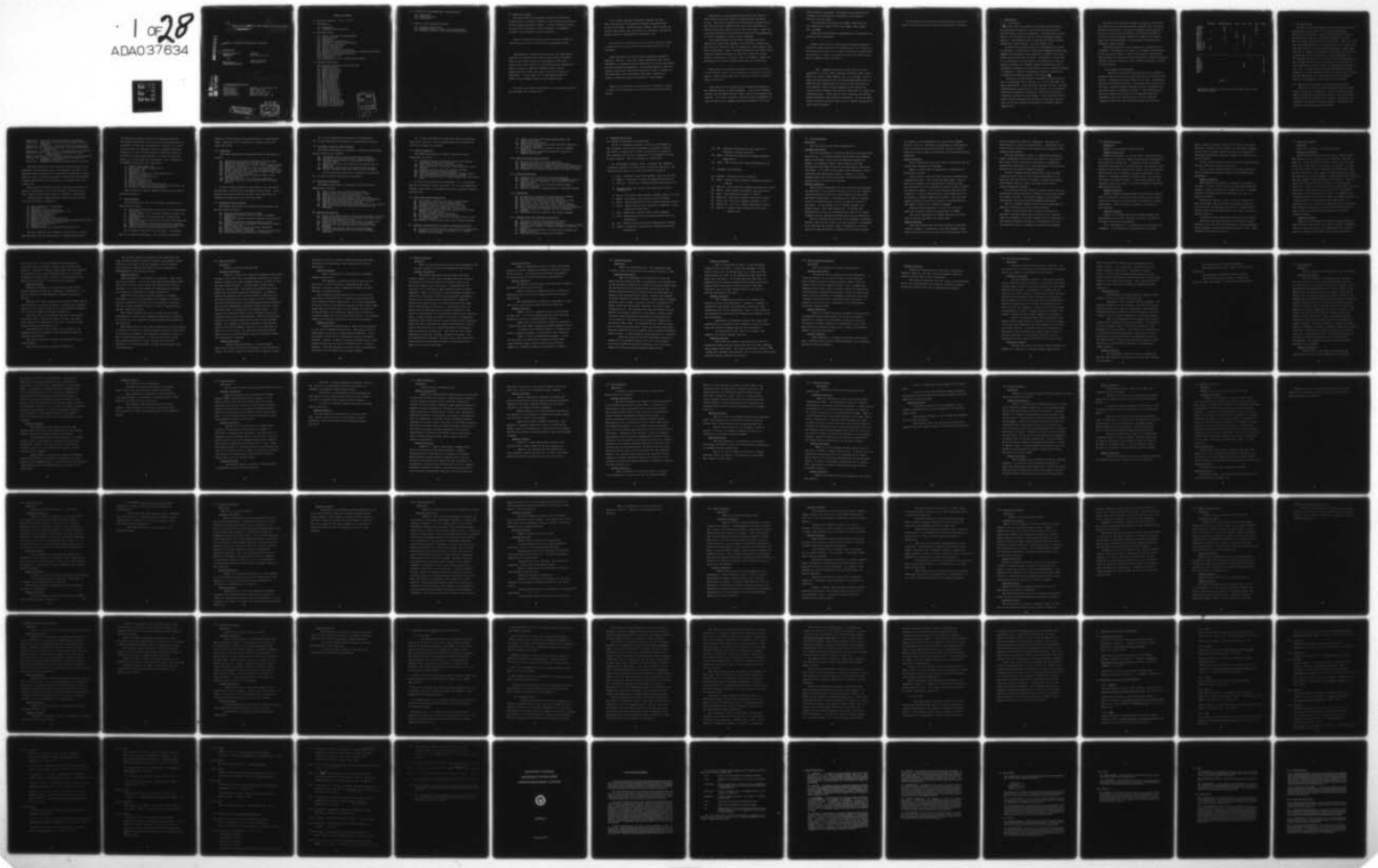
AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

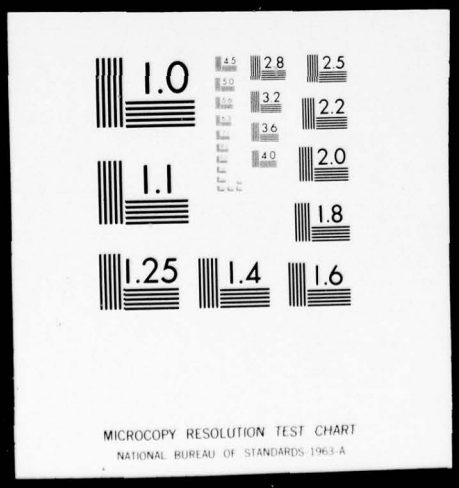
UNCLASSIFIED NL

1 OF 28
ADA037834



1 OF 28

AD A037634



AD A037634

(6) Report to the ~~HOLWG~~ High Order Language Working Group (HOLWG),

(11) 14 ~~January 14,~~ 1977

(12) 2398 p.

410122

# LANGUAGE EVALUATION COORDINATING COMMITTEE,

(10) Prepared by:


| | |
|---|---|
| Serafino Amoroso, (Chairman) | CENTACS Fort Monmouth, NJ |
| and | |
| Peter Wegner, (Technical Advisor) | Brown University Providence, RI |


2398p

(21) 29 reports in one

AD No. DDC FILE COPY

and with assistance from:


| | |
|---|---|
| Derek Morris, | CENTACS, Fort Monmouth, NJ |
| Douglas White | RADC, Rome, NY |
| Warren Loper | NELC, San Diego, CA |
| Lloyd Campbell | BRL, Aberdeen, MD |
| Calvin Showalter | NAVAIR, Washington, DC |


DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DDC
RECEIVED
APR 1 1977
A

410 122

## Table of Contents


0. Executive Summary AD A037635

1. Introduction

   1.1 Background
   1.2 Languages and Contractors

2. The Requirements

   2.0 General and Specific Requirements
   2.1 Data and Types
   2.2 Operations
   2.3 Expressions and Parameters
   2.4 Variables, Literals and Constants
   2.5 Definition Facilities
   2.6 Scopes and Libraries
   2.7 Control Structures
   2.8 Syntax and Comment Conventions
   2.9 Defaults, Conditional Compilation, Language Restrictions
   2.10 Efficient Object Representations
   2.11 Program Environment
   2.12 Translators
   2.13 Language Definition, Standards and Support

3. Language Evaluations

   3.0 Summary of Languages and Evaluations
   3.1 PL/I Evaluation
   3.2 Pascal Evaluation
   3.3 ALGOL 68 Evaluation
   3.4 HAL/S Evaluation
   3.5 PEARL Evaluation
   3.6 SPL/I Evaluation
   3.7 PDL/2 Evaluation
   3.8 LTR Evaluation
   3.9 CS-4 Evaluation
   3.10 LIS Evaluation
   3.11 Euclid Evaluation
   3.12 ECL Evaluation
   3.13 Moral Evaluation
   3.14 RTL/2 Evaluation
   3.15 FORTRAN Evaluation
   3.16 COBOL Evaluation
   3.17 ALGOL 60 Evaluation
   3.18 TACPOL Evaluation
   3.19 CMS-2 Evaluation
   3.20 SIMULA 67 Evaluation
   3.21 Jovial J3B Evaluation
   3.22 Jovial J73 Evaluation
   3.23 Coral 66 Evaluation



ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A 23

4. Discussion, Recommendations, and Conclusions

   4.1 Discussion
   4.2 Recommendations
   4.3 Conclusions

5. Guide to the Supporting Documents

   5.1 Language Requirements
   5.2 Documents used as a basis for the Evaluation
   5.3 Documents produced by the Language Evaluations

## 0. Executive Summary

The objectives of the language evaluation coordinating committee are to evaluate, summarize, and structure the findings of the language evaluation reports. In this executive summary, we present the essentials of these findings. An expanded version of this summary is found in Section 4.

*Among all the languages considered none was found that satisfies the requirements so well that it could be adopted as the Common Language.*

Each feature or capability mentioned in the requirements document can essentially be found in some existing language, hence, some minimal collection of languages exists which collectively would contain all these features and capabilities. However, there are important embedded computer system applications that could make good use of all the major requirements. In fact, most of the requirements would be useful in any embedded computer system applications.

*All evaluators felt that the development of a single language satisfying the requirements was a desirable goal.*

It is clearly possible to design a language by brute force containing all the technical features and capabilities of the requirements. Problems arise, however, when one adds the general requirements such as simplicity, uniformity, reliability, design integrity, implementation efficiency, etc.

*The concensus of the evaluators was that it would be possible to produce a language within the current state of the art meeting essentially all the requirements.*

Some evaluators felt that certain requirements should be modified. However, it was felt almost unanimously that the development of a language meeting essentially all of the requirements was both feasible and desirable. The precise degree of trade-off between potentially conflicting requirements (such as simplicity and generality) can be determined only after a substantial amount of additional work on the design of the language.

*Almost all the evaluators felt that the process of designing a language to satisfy all the requirements should start from some carefully chosen base language.*

Working from a base language would reduce the amount of work required and would reduce the opportunities for making errors. There was no consensus as to which base language to use, but every evaluator indicated that some of the languages considered were more suitable for this role than others. There was unanimous agreement among evaluators concerning the unsuitability of certain languages to serve as base languages.

Even though almost all felt that a modification effort should start from a base language, all felt that a design team must have the freedom to make any changes to a base language they feel is warranted. Hence, the specification for the design effort should be carefully written to avoid any artificial or undesirable restrictions on a design team. For example, upward compatibility with a base language should not be a restriction.

*Without exception, the following languages were found by the evaluators to be inappropriate to serve as base languages for a development of the Common Language: FORTRAN, COBOL, TACPOL, CMS-II, JOVIAL J73, JOVIAL J3B, SIMULA 67, ALGOL 60, and CORAL 66.*

This should not be interpreted as a statement concerning the technical merits of these languages. Some of the languages on this list are among the most widely used of the languages considered. It is only a statement concerning their suitability to serve as bases for a language modification relative to all the

other languages considered. The reasons for rejecting these languages are varied and are discussed in some detail in Section 3 of this report.

The languages that remain are: PASCAL, ALGOL 68, PL/I, LIS, EUCLID, CS-4, PDL/2, RTL-2, LTR, PEARL, SPL/I, HAL/S, ECL, and MORAL.

We believe the following recommendation is consistent with almost all the evaluations.

*Proposals should be solicited from appropriate language designers for modification efforts using any of the languages PASCAL, PL/I, or ALGOL 68 as base languages from which to start. These efforts should be directed toward the production of a language that satisfies the DoD set of language requirements for embedded computer applications.*

The languages that have not been found to be inappropriate to serve as base languages are HAL/S, PEARL, SPL/I, PDL/2, LTR, RTL/2, MORAL, EUCLID, LIS, CS-4, and ECL. Most of these are modifications of one of the languages PASCAL, PL/I, or ALGOL 68 for an application area close to that with which we are concerned. Many of these languages have features that satisfy certain important DoD language requirements in especially interesting ways. Hence, many of these languages are relevant design experiences that should be considered. The design teams should feel free to make as much use as is deemed appropriate of any of these languages.

*At some appropriate time some choice should be made among these design efforts to determine which are most worthy of being continued to completion.*

# 1. INTRODUCTION

## 1.1 Background

The high costs for software for systems developed within the Department of Defense is receiving increased attention from the highest levels of management. The major part of these costs is for software for what are called "embedded computer systems." Such systems would include tactical weapon systems, command and control systems, avionics systems, etc.

As part of the overall process of investigating the costs of software, in January 1975 a High Order Language Working Group (HOLWG) was chartered by the Department of Defense with representatives from the three services. The purpose of this group is to investigate the requirements and specifications for programming languages for embedded computer applications and to recommend the adoption or implementation of the necessary language or languages to achieve an appropriate degree of commonality of programming language usage in the services.

The first task undertaken by the HOLWG was to formulate a set of requirements for a language, or a set of languages, for these applications. This task which involved the user, research, and development organizations in the services, and the general research and industrial communities, resulted in Jan.1976 in a document informally called "TINMAN." This document which involved several iterations was believed at the time to be the final set of DoD language requirements for embedded computer applications.

The HOLWG then initiated a number of studies to investigate how closely certain existing standard languages came to satisfying these requirements. Besides the language studies sponsored by the services through the HOLWG, several other evaluations of other existing languages against the TINMAN set of requirements were also volunteered. Some of these by organizations outside the United States.

The present report has been prepared by a subcommittee of the HOLWG. Its purpose is to report to the HOLWG a consolidation of the evaluation studies. Based on the results reported from these studies the subcommittee has attempted to resolve differences, identify consensus positions, and to determine the basic findings of these studies.

### 1.2 Languages and Contractors

The task of evaluating languages against the TINMAN requirements was carried out by six contractors, two chosen by each service. Softech and CSC were chosen by the Army. Intermetrics and RLG were chosen by the Navy. IBM and SAI were chosen by the Air Force. The 23 evaluated languages and the contractors who evaluated them are indicated in figure 1. The languages above the line represent the initial set of languages chosen for evaluation, while the languages below the line represent languages added after the evaluation process was initiated.

| | SOFTECH | INTERMETRICS | RLG | CSC | SAI | IBM | OTHER |
|---|---|---|---|---|---|---|---|
| FORTRAN | | X | | | | X | |
| COBOL | | X | | | | X | |
| PL/I | | X | | | | X* | |
| TACPOL | X | X | X | | | | |
| HAL/S | | | | | | X | |
| CMS-2 | | | X | X | | | |
| CS-4 | | X | X | X | X | | |
| J-3B | X | | | X | | | |
| J-73 | | X | | X | | | |
| ALGOL 60 | | | | | X | | |
| CORAL 66 | | | X | X | | | X |
| ALGOL 68 | X | | | | X | | |
| SIMULA 67 | X | | | | | | |
| PASCAL | X | | X | | | | |
| LIS | | | | | X | | X |
| LTR | | | | | | | X |
| RTL/2 | | | | | | | X |
| PEARL | | | | X | | | X |
| SPL/I | | | | X | | | |
| EUCLID | | | | X | | | |
| MORAL | | | | | | | X |
| ECL | | | | | | | X |
| PDL/2 | | | | | | | X |

Figure 1.

*The IBM evaluation of PL/I was not done under the Air Force sponsored contract.

## 2. The Requirements

In order to make this document more self contained we shall briefly describe the set of DoD language requirements used for the evaluations. The brief description of each requirement given below is necessarily incomplete but does give an idea of the nature of each requirement. From Jan. 1976 when the TINMAN document first appeared until the present time, many comments and critiques have been prepared. These together with a workshop on the TINMAN requirements held at Cornell in late Sept. and the intensive use of the requirements in the language evaluations, has led to a better understanding of how the requirements should be formulated, and resulted in a new requirements document called the IRONMAN which appeared in Jan. 1977. In this document the DoD requirements have been organized in a very different fashion from that found in TINMAN, but the requirements are sufficiently similar in substance that the recommendations given here are not affected.

### 2.0 Organization of General and Specific Requirements

There are two levels of DoD requirements which we shall refer to as general and specific. General requirements are global language characteristics related to the overall design objectives for the language while specific requirements are concerned with specific language features. The general requirements may be summarized as follows:

Simplicity: avoid unnecessary generality or complexity
Reliability: properties to aid in program safety and error detection
Readability: readability is more important than writability
Maintainability: emphasize modularity, clarity, documentation, few defaults
Efficiency: no sacrifice of run time efficiency for generality
Implementability: state of the art features with known implementation
Machine Independence: well defined interface to object machines
Portability: adaptable to different object machines and different applications
Definition: unambiguous, complete, understandable definition

The above objectives reflect the fact that the costs of the operations and maintenance part of the software life-cycle for embedded computer applications are generally considerably greater than the costs of program development, and the fact that embedded computer applications are often subject to stringent real-time constraints.

These general requirements serve to motivate the choice of specific requirements, and in many instances constrain the way that a specific requirement should be realized or implemented.

The 98 specific TINMAN language requirements are grouped into the following 13 categories:

A. Data and Types (7 requirements)
B. Operations (11)
C. Expressions and Parameters (9)
D. Variables, Literals and Constants (6)
E. Definition Facilities (8)
F. Scopes and Libraries (7)
G. Control Structures (8)
H. Syntax and Comment Conventions (10)
I. Defaults, Conditional Compilation and Language Restrictions (7
J. Efficient Object Representations (5)
K. Program Environment (5)
L. Translators (9)
M. Language Definition, Standards and Control (6)

The above organization of language characteristics is somewhat different from the organization of language descriptions

in programming language manuals and language definitions. The IRONMAN requirements specification has reorganized the requirements so that they correspond more closely to the order of presentation of language features in language definition documents, but has not substantially altered the overall requirements for the common language. In order to bring the reader up to date concerning requirements, the 13 categories of IRONMAN requirements are listed below:

1. General Requirements (8)
2. Syntax and Comment Conventions (9)
3. Data Types (3)
4. Expressions (7)
5. Constants, Variables, and Declarations (7)
6. Control Structures (5)
7. Functions and Procedures (9)
8. Input-Output (5)
9. Parallel Processing (6)
10. Exception Handling (6)
11. Machine Dependent Specifications (6)
12. Library, Separate Compilation, Generic Definitions (4)
13. Standards, Translation and Support (7)

The 98 individual specific TINMAN requirements are briefly characterized below.

## 2.1 Data and Types

The requirements in the "Data and Types" category may be summarized as follows:

- A1. Date types determinable at compile time and unalterable at run time.
- A2. Integer, fixed, float, Boolean, character, array and record types.
- A3. Precision specs for floating point arithmetic and variables.
- A4. Exact fixed point numbers with user specified range and fractional part.
- A5. Character sets with user defined collating sequence.
- A6. Arrays with static lower bound and dynamic upper bound.
- A7. Variant records fully discriminated at run time.

A1 is a general requirement on data types. A2 specifies the set of required data types. The remaining requirements

specify in greater detail the characteristics of required data types. Requirement A7 is intended as a substitute for the union data types.

## 2.2 Operations

The TINMAN requirements on operations may be summarized as follows:

- B1. Assignment and reference operations for data types
- B2. Equivalence operator for all data types
- B3. Relational operations for numeric and enumeration types
- B4. Arithmetic operations +, -, *, /, ÷ , ↑ , unary minus
- B5. Truncation and rounding of least significant digits
- B6. Boolean operators and, or, not, xor, short circuit mode
- B7. Direct assignment for comformable composite data types
- B8. No implicit type conversion
- B9. No conversion required for numeric ranges, range checking optional
- B10. I/O operations for files, channels, terminals
- B11. Power set operations (logical operations on Boolean vectors).

B1 and B2 specify operations applicable to all data types. B8 specifies a restriction on conversion between types. The remaining requirements indicate specific types required by the language and properties of some of these types.

## 2.3 Expressions and Parameters

The TINMAN requirements on expressions and parameters may be summarized as follows:

- C1. Side effects evaluated left to right
- C2. Readable expressions with few levels of operator precedence
- C3. Expressions permitted whenever constants and variables allowed
- C4. Constant expressions evaluated before run time.
- C5. Consistent rules for parameters of procedures, arrays, declarations, etc.
- C6. Type agreement of formal and actual parameters
- C7. Classes of formal parameters
- C8. Optional parameter attributes in procedure declaration
- C9. Procedures with variable number of parameters

C1 - C4 are concerned with properties of expressions.

C5 - C9 are concerned with parameters of procedures and arrays.

## 2.4 Variables, Literals, and Constants

The TINMAN requirements on variables, literals and parameters may be summarized as follows:

- D1. Identifiers with constant values may be defined
- D2. Constants will have some value in programs and data
- D3. Declared variables may be initialized. No default initial values
- D4. Range and step size for fixed point variables must be specified
- D5. Arrays and records may have components of any type
- D6. Pointer variables must be as safe as other variables

D1 and D2 specify properties of literals and constants.

D3 - D6 specify certain properties of variables.

## 2.5 Definition Facilities

The TINMAN requirements on definition facilities may be summarized as follows:

- E1. Users will be able to define new data types
- E2. Defined types will behave like built-in types
- E3. There will be no default declarations
- E4. Operations will be extendable to new data types
- E5. Type definitions do not automatically inherit operations
- E6. New types may be defined by enumeration, Cartesian product, discriminated union, power set
- E7. Type definition by free union and subsetting is not desired
- E8. Type initialization and finalization procedures are definable

## 2.6 Scopes and Libraries

- F1. Distinction between scope of allocation and scope of access
- F2. Access to identifiers can be limited both at their point of definition and point of call
- F3. Scope of identifiers will be determined at compile time
- F4. Libraries will be supported and easily accessible
- F5. Libraries will not exclude routines written in other languages
- F6. Libraries and compools will be indistinguishable
- F7. Standard library definitions for machine dependent interfaces

F1 - F3 are concerned with scopes and rules for accessing identifiers while F4 - F7 are concerned with the interface between the language and libraries.

## 2.7 Control Structures

The TINMAN requirements on control structures may be summarized as follows:

- G1. Structured control mechanisms, parallel processing, exception and interrupt handling
- G2. Go-to only within most local access scope
- G3. Fully partitioned if-then-else, case statement, Zahn's device
- G4. Iterative control with local control variable
- G5. Recursive and non-recursive routines
- G6. Parallel processes, synchronization, critical regions
- G7. User defined parameterized exception handling
- G8. Real and simulated time, relative priorities, synchronization

G1 lists the desired control mechanisms. G2 - G5 indicate desired conventional control structures. G6, G7, G8 respectively indicate requirements for parallel processing, exception handling and real-time.

## 2.8 Syntax and Comment Conventions

- H1. Free format, statement delimiter, easily parsed
- H2. No modification of source language syntax
- H3. Language definable in 64-character ASCII set
- H4. Formation rules for identifiers and literals
- H5. No continuation of lexical units across lines
- H6. Keywords will be few, reserved, informative, not confusable with identifiers
- H7. Uniform readable comment convention
- H8. No unmatched parenthesis are permitted
- H9. No language imposed distinction between function calls and data selection
- H10. Symbols in same context cannot have different meaning

## 2.9 Defaults, Conditional Compilation, Language Restriction

- I1. No undefined defaults which affect result of computation
- I2. Defaults which optimize implementation of language features are encouraged

I3. Compile time variables which specify object computer environment
I4. Conditional compilation
I5. Simple base language which allows efficient definition of complete language
I6. Translator restrictions should be part of language definition
I7. Object machine restrictions should not be part of language definitions

## 2.10 Efficient Object Representation

J1. No run-time costs for unused generality
J2. Language design should allow safe optimizations
J3. Encapsulated access to hardware facilities and machine code
J4. Object representation of data structures can be specified
J5. Programmer can specify routine calls to be open or closed

## 2.11 Program Environment

K1. Language will not require an operating system
K2. Language will support integration of independent modules
K3. Linkers, loaders, debuggers, and other systems software available
K4. Documentation, editing, testing and other support software available
K5. Optional assertions, debugging specs, measurement probes

## 2.12 Translators

L1. No supersets. Features not permitted are forbidden
L2. No subset implementations will be allowed
L3. User control of optimization and compile time costs
L4. Translators for a variety of object machines
L5. Translator is not required to run on object machine
L6. Syntax checking but not error correction by translator
L7. Error diagnostics specified as part of language definition
L8. Internal translator structure not part of language standard
L9. Translators will be written in the source language

## 2.13 Language Definition Standards and Control

M1. Individual features must be state of the art
M2. Unambiguous and clear language definition
M3. Tutorial and reference documents, defined by abstract computer
M4. Configurations control to ensure translators conform to standard
M5. Support agent responsible for maintaining language and support software
M6. Standards and support agents for libraries

## 3. Language Evaluations

### 3.0 Summary of Languages and Evaluations

Of the 23 languages considered, three were recommended as base languages, eleven were recommended as relevant to the design effort or deserving of further consideration as potential base languages and nine were regarded as not acceptable to serve as base languages. The 23 languages are listed below.

A: Recommended Languages (These languages each represent a different synthesis of large amount of previous experience, and constitute the nucleus of a family of derived languages).

1. PL/I: Includes concepts from FORTRAN, ALGOL 60 and COBOL
2. PASCAL: A successor of ALGOL 60 emphasizing simplicity
3. ALGOL 68: A successor of ALGOL 60 emphasizing generality

B: Languages which are Relevant and Deserving of Further Consideration

4. HAL/S: PL/I based, NASA Language, strongly typed, real-time
5. PEARL: PL/I-based, German process control language
6. SPL/I: PASCAL-based NRL real-time signal processing language
7. PDL/2: PASCAL with parallel processing, independent module facilities, Texas Instr.
8. LTR: PASCAL-based official French common language
9. CS-4: PASCAL-based real-time with extension facilities, Intermetrics
10. LIS: PASCAL-based French system implementation language
11. EUCLID: PASCAL-based experimental language emphasizing verification

12. ECL: Extensible language with good support environment, Harvard University

13. MORAL: New British language for embedded computer applications

14. RTL/2: Real-time British language developed at ICI

C: Languages Not Acceptable

15. FORTRAN: Developed by IBM in 1954-58

16. COBOL: Business data processing language developed in 1959-61

17. ALGOL 60: Block structure language developed in 1957-60

18. TACPOL: Army language developed in the late 1960's

19. CMS-2: Navy language developed in 1966-69

20. SIMULA 67: Simulation language developed in Norway

21. JOVIAL J3B: Air Force language developed in 1972

22. JOVIAL J73: Air Force language developed in 1969-73

23. CORAL 66: British common language for real-time applications

## 3.1 PL/I Evaluation:

### Evaluators

PL/I was evaluated by IBM and Intermetrics.

### Language Description

PL/I was developed in the period 1962-66 by an IBM-sponsored design group. It includes concepts and features from FORTRAN, ALGOL 60 and COBOL, taking over its expression syntax from FORTRAN, its statement syntax, block structure and type declarations from ALGOL 60 and its data description facilities from COBOL. It is rich in data types, data attributes, control structures and input-output features. However, it is not strongly typed and lacks extensibility, parallel processing, synchronization and real-time features.

### Overall Evaluation

IBM strongly recommended PL/I as a potential base language, and presented a fairly detailed discussion of how the language could be modified to meet what they interpret as essentially the TINMAN requirements. Intermetrics felt that the language should be rejected because it lacks too many of the DoD requirements. It was designed for a different purpose (expressive power rather than readability, reliability and modifiability) and it was developed before many of the modern language features required by TINMAN were properly understood.

PL/I is the most controversial of the languages considered. However, since the use of PL/I as a base language is recommended by an important and credible group of designers, we believe that this group should have an opportunity to further explore

its approach to the development of a common DoD language. Moreover, there is a great deal of experience in the development of subsets and derivatives of PL/I and some of these derivatives such as HAL/S and PEARL may well be suitable design experience.

Positive Features

PL/I is more widely used by an order of magnitude than any of the other proposed base languages.

There is a great deal of experience in subsetting and modification of PL/I.

A great deal of care and effort has been lavished on the language definition. The defining document BASIS 1 has recently been adopted as an American PL/I standard by ANSI. The availability of this standard may make it easier to define the modified language. The work on language definition has resulted in a better understanding of the language which will be helpful in language modification efforts. The tools and support software for PL/I may be useful in developing tools and support software for a modified language.

IBM has already considered in some detail how PL/I should be modified to meet the requirements of TINMAN.

PL/I has an advantage over PASCAL and ALGOL 68 in the area of external procedures and common data.

Negative Features

PL/I was designed with different objectives from those stated in TINMAN. In particular, PL/I was designed to promote power of expression and ease of writing programs, while

the DoD requirements emphasize readability, modifiability, maintainability, security, and efficiency. Achievement of the latter kinds of objectives would require large changes in the language specification.

PL/I was developed too early to incorporate advances of design technology of the mid and late 1960's in the areas of data types, control structures, extensibility, modularity, programming methodology, etc. Such developments have not been reflected in the recently adopted standard.

The PL/I language design suffers from a lack of design integrity since it has not been entirely successful in integrating the wide variety of language concepts and features from FORTRAN, ALGOL 60, COBOL, and other sources into a single homogeneous language.

PL/I provides too much freedom of expression for the user, attempting to assign meanings to programs in cases which should be treated as errors. The current emphasis is to introduce restrictions that enforce good programming methodology.

PL/I is a complex language for which the consequences of addition and deletion cannot be easily foreseen.

There is some concern that it would be difficult to evaluate a language design starting from PL/I as a base because the complexity of the base language and the magnitude of the language changes would make it very difficult for a design evaluation team to evaluate the proposed new language.

## 3.2 PASCAL Evaluation:

### Evaluators

PASCAL was evaluated by Softech and RLG.

### Language Description

PASCAL was developed in the late 1960's by Niklaus Wirth as a successor to ALGOL 60 emphasizing simplicity (as opposed to the generality of ALGOL 68). It provides richer declarative facilities than ALGOL 60, including records (structures), files, pointers, scalar types (enumeration types), sets and programmer-defined types, but is otherwise as simple as possible. For example, it excludes parameters called by name, own variables and arrays with dynamic bounds. It has strong (compile time checkable) typing and control structures designed to encourage good programming style.

### Overall Evaluation

Softech recommends PASCAL because relatively few features need to be deleted, and because those features which it does have are well-designed and very much in the spirit of the DoD requirements. RLG also feels that PASCAL is suitable as a base language.

### Positive Features

PASCAL is a strongly typed well designed language with extensibility and control structure facilities very much in the spirit of the DoD requirements.

It is very popular as a starting point for designing new languages. It has served as a starting point for larger

general purpose languages, such as CS-4 and LIS, for signal processing languages, such as SPL/I, for real-time languages, such as PDL/2, and LTR, and for experimental languages for concurrent programming (concurrent PASCAL) and program verification (EUCLID).

It has a formal axiomatic definition which has been widely used as a basis for program verification.

It is a simple language which can be easily understood both by programming language designers and by applications programmers.

Negative Features

The experience with PASCAL is largely in a research environment. It has not been widely used for large production programs. However, extensions of PASCAL for use in embedded computer environments have been developed in a number of places (see PDL/2).

Because the language is small, a lot of features will have to be added, such as parallel processing, real-time, error handling, and precision facilities. In adding these features there ary many opportunities for errors and loss of design integrity.

It does not permit independent program and data modules.

There are some slight problems with existing language features such as the fact that dimensions of array parameters must currently be bound at procedure declaration time. These problems can easily be fixed up.

## 3.3 ALGOL 68 Evaluation

### Evaluators

ALGOL 68 was evaluated by Softech and SAI

### Language Description

ALGOL 68 was developed during the period 1963-69 by the IFIP working group 2.1 (WG2.1) as the "official" successor to ALGOL 60. It has the block structure of ALGOL 60 with a much richer set of data types (called modes in ALGOL 68). It has an infinite number of modes obtainable from eight basic modes by mode constructors such as ref for specifying pointers, proc for specifying procedures, struct for specifying record structures, and long, short for specifying numerical precision. There are complex but consistent rules for implicit mode conversion (coercion). There is a fairly rich set of input-output (transput) operations, a well designed set of facilities for parallel processing, but no real-time facilities. The language has a "standard prelude" which defines convenient language extensions, a "library prelude" for defining library facilities and a "system prelude" for defining the operating system environment. The language definition uses an expressive but difficult-to-master syntactic mechanism called W-grammars to express program structure.

### Overall Evaluation

ALGOL 68 is recommended as a potential base language by both Softech and SAI because its design integrity, consistency and clean definition make it more appropriate as a starting point for language design than a base language designed in

an ad hoc manner. It is the language most consistently recommended by its evaluators among all the candidate base languages. However, we talked to many people who were skeptical of using ALGOL 68 as a base language because it is not well understood or widely used in the U.S.A. and because there are apparently difficulties in implementing the language.

Positive Features

ALGOL 68 is designed in an exceptionally clean manner. Because language features are "orthogonal" (independent), it is easy to determine the consequences of adding and deleting features.

ALGOL 68 is a large language which already contains many of the more difficult required features. Features not in ALGOL 68 would have to be added to many of the other candidate languages too, but can be added more cleanly in ALGOL 68.

SAI states that one immediately sees how to add abstract data types, sets, fixed point numbers, or other data types, while reducing the generality of the go-to, eliminating flex arrays and certain operations, etc.

The language definition reflects the cleanness of the language design and is helpful in determining the effect of language modifications.

The control structures of ALGOL 68 encourage writing of clean programs.

The strong typing aids in program reliability.

The reference concept with explicit local generators and heap generators appears to be well suited to the dynamic storage allocation philosophy of the DoD language, but will require some additional language-level specification to explicitly distinguish all stack and heap storage.

Negative Features

ALGOL 68 appears to be difficult to implement. Apart from the British implementation at the Royal Radar Establishment (RRE), there does not seem to be a widely used implementation of the language. It is not clear whether the implementation problems are intrinsic or simply due to a lack of resources.

ALGOL 68 is less widely used than competitor languages like PASCAL. It is not clear whether the language is intrinsically too "difficult" for applications programmers or whether the lack of user enthusiasm is due to a lack of implementations and poor expository manuals.

The current language definition is difficult for the uninitiated but is said to be an advantage rather than a hindrance to its use as a base language once the forbidding terminology has been mastered.

The complex coercion rules make it difficult for the programmer to determine the effect of certain program constructs and will have to be replaced by explicit mode conversion functions in order to meet the TINMAN requirements. It seems a pity to throw away this language capability, since it is not clear that mode conversion in a cleanly designed, strongly typed language leads to unreliability.

## 3.4 HAL/S Evaluation

### Evaluators

HAL/S has been evaluated by IBM.

### Language Description

HAL/S is a PL/I-style language developed by Intermetrics for NASA in the early 1970's and designed especially for use on airborne computers and other embedded systems in connection with the space shuttle program. Its data types include arrays, structures and pointers. Non-recursive procedures may be declared but not passed as procedure parameters. It requires strong type specifications for both declared identifiers and procedure parameters, has a wide range of control structures for structured programming, multiprogramming and error handling, and provides independently compilable program and data modules which facilitate the construction on large programs. It has a real-time processing facility with a flexible facility for scheduled processes in a process queue. Execution and storage allocation for scheduled processes is left to the operating system. The language is carefully designed so that storage requirements for all programs and tasks can be determined at process initiation time. The operating system must decide when initiating a process whether there is enough free memory for that process to operate.

### Overall Evaluation

IBM has recommended HAL/S as a base language. HAL/S does not provide recursive procedures, multi-dimensional arrays with dynamic bounds and other forms of dynamic storage

allocation in order to achieve efficient runtime code while providing multiprogramming, error handling and real-time facilities.

Positive Features

There is experience in using HAL/S for embedded computer applications.

The language is reasonably clean in spite of the fact that it contains pointers, multiprogramming, error handling, real-time control, independent modules and other difficult to handle features.

The language has proved sufficiently successful to be under active consideration for a number of new military and civil embedded computer applications. It is being considered by the Air Force as the on-board language for the interim upper-stage (IVS), by Collins Radio as the language for the AP-101 on-board guidance and navigation computer and by Boeing as a standard language for civil avionics applications. The language manuals and language specification are clear and adequate.

Negative Features

Substantial modifications to HAL/S would be required in the area of language extensibility. There are no user defined types nor any kind of encapsulation mechanism. It has no recursive procedures or other dynamic storage allocation features. Finally, a number of features of HAL/S used to solve parallel processing and real-time problems may be in need of updating. They were designed in the late 1960's and should be re-examined in the light of more recent research.

## 3.5 PEARL Evaluation

### Evaluators

PEARL was evaluated by CSC and was also evaluated by GMD in Bonn, Germany. It was also evaluated by John Williams.

### Language Description

The Process and Experiment Automation Real-Time Language (PEARL) is a PL/I style language which was developed in Germany in the period 1968-73 by a working group of experts from industry and universities. Implementations are available or are underway on eight different process control computers. PEARL is a procedure oriented language designed for reliability, efficiency, portability and machine independence in real-time and process control applications. PEARL has standard data types as well as clock and duration data types, but no mode definition facilities. Procedures are not regarded as data types and must be declared reentrant if they are to be interruptable and executable by several tasks simultaneously. A PEARL program can consist of a series of separately compilable modules each having a system division with information about the system configuration, device interconnection, process signals and terminals and a problem description with module, task and procedure declarations. Tasks may have four states referred to as running, runnable, suspended or dormant, may be synchronized by semaphore variables, may be scheduled using clock and duration data types and may be interrupted by programmed or real-time interrupts.

Overall Evaluation

PEARL is comparable to HAL/S in being a PL/I-based procedure oriented language with parallel processing and real-time facilities. Its portability and machine independence is impressive and it certainly deserves consideration.

Positive Features

It contains well-developed real-time process control and parallel processing facilities.

There has been some experience both in language implementation and in using the language for process control applications.

The language-system interface is specified in a system division and appears to have been well worked out.

Negative Features

PEARL, like PL/I, appears to lack a clear concept of data type. It is not strongly typed and lacks data-type definition facilities and extensibility features.

PEARL does not have pointers, recursive procedures, a fixed point data type or adequate exception handling features.

It lacks a well defined interface to routines compiled in assembly language or other higher level languages.

PEARL is still an experimental language. It has not been used as an operational language for any really large embedded computer applications and has not been adopted as a standard language by government or industry.

## 3.6 SPL/I Evaluation

### Evaluators

SPL/I was evaluated by CSC. The discussion below is based in part on discussions with Richard Harrington of NRL.

### Language Description

SPL/I was developed in the period 1972-76 by the Naval Research Laboratory (NRL) in cooperation with Intermetrics. It was developed originally as a Signal Processing Language (SPL) for processing real-time analog inputs and performing fast Fourier transforms and other digital filtering operations for analyzing and processing these inputs. However, the language is general-purpose with block structure, strong typing, standard primitive types, arrays, structures, pointers, procedures with input, output and input-output parameters, no implicit mode conversions, etc. The language has arrays with dynamic bounds, recursive procedures and permits truly dynamic storage allocation but allows optimization when dynamic storage allocation is not required. Procedures can be separately compiled and can contain global data. Multiprocessing with synchronization can be performed using process, signal and resource data types. Real-time applications can be handled using a machine-independent clock facility. An error-handling escape mechanism is available.

SPL/I was initially used for underwater signal processing with the PROTEUS computer, but it is now available on the Navy AN/UYK-20 and AYK-14 computers, and is being considered for non-Navy embedded computer applications.

Overall Evaluation

SPL/I is comparable to HAL/S. It was developed slightly later than HAL/S. It has the advantage of being able to make use of the design experience of HAL/S and the disadvantage that there is less operational experience with SPL/I than with HAL/S. It is more ambitious than HAL/S in handling parallel processing and real-time applications with dynamic rather than static allocation within individual processes, and is in this respect closer to the spirit of the DoD requirements than HAL/S.

Positive Features

SPL/I substantially satisfies the parallel processing and real-time requirements. Moreover, it appears to satisfy the requirements of simplicity, reliability, maintainability and machine independence. Thus, it must certainly be considered by any design team modifying any language to meet the requirements.

Experiments with PROTEUS indicate that object code produced by the Intermetrics SPL/I compiler is only 10% slower than good assembly language code for the same program.

In the opinion of CSC, SPL/I is a cleaner, more readable and simpler language than CS-4.

Negative Features

SPL/I does not include a type definition facility, enumeration types, precision specifications, and other important DoD language requirements. The control structures of SPL/I, including both sequential and parallel, are not the best ones possible and their designs should be re-examined.

## 3.7 PDL/2 Language Evaluation

### Evaluators

PDL/2 was evaluated by Texas Instruments Inc.

### Language Description

The Process Design Language PDL/2 was developed by Texas Instruments Inc. in the mid 1970's for the Ballistic Missile Defense Advanced Technology Center (BMD) to meet the needs of BMD real-time software design. It is a PASCAL-based language which extends PASCAL by adding tasking and synchronization primitives, variable length arrays, vector and array operations, assertion statements, independent compilation and common variables, code generation, macro substitution facilities. A compiler for PDL/2 has been implemented on the BMD advanced scientific computer (ASC).

### Overall Evaluation

PDL/2 is a direct extension of PASCAL in the direction of embedded computer applications, and should be carefully studied as an example of an attempt to do precisely the kind of design by language modification recommended in this report. PDL/2 should be carefully looked at by a design team modifying PASCAL to meet the requirements.

### Positive Features

PDL/2 represents a language modification effort which goes a substantial way towards converting PASCAL into a language meeting the DoD requirements.

Negative Features

Substantial modifications in the areas of precision, exception handling, and other areas are needed to make the language conform with the DoD requirements.

The specific mechanisms used for tasking, synchronization, global data specification, etc. should be carefully analyzed to determine whether they can be significantly improved.

## 3.8 LTR Language Evaluation

### Evaluators

LTR was evaluated by Alan Demers at Cornell. Conversations with Pierre Parayre of the French Naval Programming Center were helpful in developing this evaluation.

### Language Description

LTR was designed in France in 1968 for the realization of multitask real-time systems. It is block structured, strongly typed, and has a rich and adequate set of data types and control structures. In addition, it includes the concepts of task, event, resource, system time, delay, interrupt real-time I/O, and primitives for handling these entities. LTR has proved to be well suited for modular programming and structures programming and is almost entirely portable between 16 and 32 bit computers. Since 1973 LTR has been used for programming a number of complex operational systems, and assembly language has been almost completely circumvented.

LTR is the official French common language for embedded computer applications in the Navy, Army and Air Force. It has also been adopted as the common language for civilian air traffic control. It has been used as the language for developing a Naval Technical Data System (NTDS) similar to that for which CMS-2 is being used in the US, and for a battlefield tactical system called ATTILA (similar to TOS).

### Overall Evaluation

LTR has been widely used by the French Department of Defense as a language for embedded computer applications,

and clearly satisfies many of the DoD requirements for a common language. It is, therefore, very relevant to the common language design effort. According to Parayre, there is an able French language design group at CII who would be eager and willing to develop and submit a preliminary design for the common DoD language. Such a design would be based on great experience with HOLs for embedded computer application.

Positive Features

The language is strongly typed and contains modern notions of control structure and modularity.

LTR appears to be a highly efficient language. LTR has well designed real-time features. The ability to create and destroy processes, process communication and scheduling, interrupt handling, and direct I/O are all present with about as high a level of machine independence as one could expect.

LTR is the only example of an apparently successful common HOL for embedded computer applications. The existence of such a language should increase our confidence in the feasibility of this project. At the same time, we should try to learn from the French experience, either by soliciting preliminary design based on LTR or by involving a member of the French group in some other way to provide inputs concerning technical and managerial experience in introducing a common language.

Negative Features

The equivalence mechanism, quality variables and the free type option for pointers allow strong type checking to be defeated.

The limitation to one-level structures.

The implementation of FOR loops with the test at the bottom.

The language is oriented to 8-byte machines.

The real-time features require a special LTR monitor so that the language is operating system dependent.

## 3.9 CS-4 Evaluation

### Evaluators

CS-4 was evaluated by Intermetrics, CSC, RLG and SAI.

### Language Description

CS-4 was designed by Intermetrics for the Navy in the mid 1970's and is not yet implemented because work on its implementation was stopped as part of the "freeze" on DoD new programming language development pending outcome of the HOLWG study. The design of CS-4 was strongly influenced by PASCAL, HAL/s and modern programming language concepts. The language has strong-type checking with few implicit conversions, powerful data structuring tools with full safety, ability to define new abstract types (although not new infix operators), support for parallel processing and exception handling, user control over precision and range of variable values, and adequate control structures. There are adequate separate compilation facilities and good facilities for escape into machine language. CS-4 has no pointers, recursive procedures, machine independent interfaces to hardware components or generic procedures. Although the language is fairly well designed, there are, in the opinion of CSC, clumsy design features in the area of parallel processing and in the wordiness of declarations. The language is large and certain features would have to be deleted.

### Overall Evaluation

Intermetrics is very positive concerning the suitability of CS-4 as a base language while RLG and CSC

are mildly positive but not enthusiastic. Intermetrics feels that CS-4 is particularly strong in the areas of reliability, maintainability, transportability and language power. CSC and RLG are concerned with the complexity and lack of implementation of CS-4. Since CS-4 represents a large and skillful design effort with similar goals to those of the current DoD effort further refinement of this design to meet the specific DoD requirements should certainly be considered. However, CS-4 is not as well defined a starting point for a new design as an already implemented language, and it may well be that designers might wish to design a CS-4 like language while using some other language as a base language.

Positive Features

The language has made extensive use of modern language design principles and includes many of the language features specified in the DoD language requirements.

The language has been specially designed to satisfy current notions of reliability, maintainability and transportability. However, this claim cannot be convincingly verified, since there is no implementation and no user experience for the language.

Strong typing, extensibility, parallel processing, precision specification and separate compilation are incorporated in the language design. It is only one of the languages considered that meets the requirements in all five of the above areas.

Negative Features

The language is not implemented.

Although the language description is generally clear, there is no really precise language definition.

The language is so large that a clean definition or an implementation is necessary to determine whether claims made for the language are in fact true.

The language lacks pointers and recursive procedures. These features could easily be introduced into the language but would considerably complicate the implementation.

## 3.10 LIS Evaluation

### Evaluators

LIS has been evaluated by SAI and Alan Demers at Cornell.

### Language Description

LIS was designed by Jean Ichbiah and the French company CII/GAM as a system implementation language. It is strongly typed, has a rich set of data types and control structures, has good facilities for independently compiled program and data modules, and specifies machine dependent features in separately defined modules called implementation parts. It has coroutines but no parallel processing, exception handling or real-time facilities.

### Overall Evaluation

LIS has most of the features of a modern block structure language and is designed to meet criteria of efficiency, reliability and modularity. The experience and ingenuity of the LIS design group is considerable, and it is quite possible that a good language meeting the DoD requirements could be designed starting from LIS as a base. Since the French approach is inevitably likely to be different from that of U.S. design groups, it might be instructive comparing the results of a French design effort with the results of design efforts by U.S. Software houses.

### Positive Features

LIS has been designed according to modern notions of efficiency, reliability and modularity.

The goals of system programming languages, such as LIS, overlap to a considerable degree with the goals of languages for embedded computer applications.

The notions of independently compiled program and data modules and of implementation parts are better worked out than in comparable other PASCAL-oriented languages.

It is one of the best designed extensions to PASCAL that is available.

Negative Features

LIS lacks precision specification, fixed point data type, operator extension, generic procedures, parallel processing, real-time facilities and exception handling facilities.

## 3.11 EUCLID Evaluation

### Evaluators

EUCLID has been evaluated by CSC.

### Language Description

EUCLID is a PASCAL-based language designed specifically to facilitate program verification. Restrictions are placed on certain language constructs which allow the compiler and verifier to make stronger inferences about language properties, thereby increasing reliability, efficiency and verifiability. These restrictions include explicit control over imported and exported identifiers in procedures and modules, and guarantees that two identifiers in the same scope (including procedure parameters) can never refer to the same or overlapping variables. Extensions of PASCAL include parametrized types, modules which can contain procedures and data components, and assertion specifications. Deletions from PASCAL include input-output, reals, multidimensional arrays, labels, gotos, and procedures as parameters.

### Overall Evaluation

EUCLID is a "small" experimental language which meets only very few of the DoD requirements. However, language designers starting from a language in the PASCAL family should seriously consider adopting some of the restrictions and extensions of EUCLID in order to enhance the reliability, efficiency and verifiability of the new language. Although enlargement of EUCLID to meet the DoD requirements would introduce considerable complexity, CSC feels that this

approach to meeting the requirements, might if done with great care, result in an acceptable common language.

Positive Features

EUCLID's approach of enhancing reliability by placing restrictions on tricky uses of procedure parameters, identifiers and other language features may well become standard in the design of future languages.

EUCLID appears to be a well-designed language and its approach to the inclusion of parametrized types, modules and assertions deserves to be studies.

EUCLID has a clear language description. This approach to language description, which was pioneered by PASCAL, should be considered as a possible norm for language descriptions produced by language design teams for the common language.

Negative Features

EUCLID is a small experimental language which satisfies only a small number of the DoD requirements.

EUCLID is not intended for the programming of large embedded computer applications. It has no real-time parallel processing or numerical precision facilities.

## 3.12 ECL Evaluation

### Evaluators

A partial evaluation of ECL was received from Stephen Squires of NSA/CSS.

### Language Description

ECL was developed by Ben Wegbreit and first described in his doctoral dissertation in June 1970. It consists of a programming language called EL/1 and a system built around this language which allows on-line interactive construction, testing, symbolic debugging and running of programs. It allows extensibility to permit flexibility during program development as well as contractability to permit optimization for production purposes. It is a block structured language with a rich set of data types (called modes) and mode definition facilities. According to the evaluator, almost all language deficiencies, such as precision specification and fixed point variables, can be added to the language by extensions. Parallel processing and exception handling requirements are said to be satisfied by the multipath control and TRAP facilities (these facilities do not appear to be in the December 1974 language manual). The language manual describes an impressive set of supportive software including compile time facilities, built in library routines and support packages for debugging, tracing, testing, metering, etc.

### Overall Evaluation

ECL is probably too rich and flexible a language to be adopted as a realistic base for the common language.

However, it has something to teach us in the areas of extensibility and language system interface facilities, and should therefore be studied by language designers using some other language as a base. EL/1 is a large well-designed language, and the possibility of starting from EL/1 and developing a language meeting the DoD requirements by language extension and contraction should not be altogether ruled out.

Positive Features

ECL is a large well designed language which satisfies many of the difficult DoD requirements and can be made to satisfy many of the remaining ones by language extension.

Because the language is well designed it might be possible to develop the required language by placing restrictions on a suitably extended language.

Negative Features

ECL was designed for an experimental interactive programming environment rather than for production programming in embedded computer applications.

ECL is too rich in extension facilities, pattern matching, control structures, non-deterministic algorithms and a number of other areas.

## 3.13 MORAL Evaluation

### Evaluators

MORAL was conducted by Software Sciences Limited (SSL).

### Language Description

MORAL was developed in 1975-76 by the Royal Radar Establishment (RRE). Its development is based on "MASCOT," an approach to the design of real-time systems. In the interests of portability it is constrained by the adoption of CORAL 66 as a target language for the implementation of MORAL. MORAL is a block structured strongly typed language with integer, fixed point, floating point, array and record (group) data types. It lacks an explicit boolean data type and precision specifications but has enumeration types and mode definition facilities. The language supports synchronization and real-time facilities but leaves the task of creating new processes to the operating systems. Exception handling is not explicitly supported but its effect can be accomplished by an interrupt mechanism.

### Overall Evaluation

MORAL is a clean language designed specifically for real-time embedded computer applications. It handles real-time requirements differently from that envisaged by the TINMAN specification, by placing more responsibility on the operating system. It meets a surprisingly large number of the requirements and should be seriously considered by design teams as a relevant language.

### Positive Features

MORAL was designed for the same purpose as the common DoD language.

It meets a surprisingly large number of the requirements.

It is one of the most recently designed languages considered and has made good use of ideas in PASCAL, ALGOL 68, CORAL 66 and other languages.

Negative Features

A prototype translator for the language was completed in September 1976, but the language has not yet been fully tested.

The real-time features of the language are dependent on its operating system.

The language design was constrained by requiring CORAL 66 to be the target language of the associated translator.

## 3.14 RTL/2 Evaluation

### Evaluators

RTL/2 was evaluated by Imperial Chemical Industries (ICI) and by Alan Demers at Cornell.

### Language Description

RTL/2 is an ALGOL-based language developed in Britain by ICI (Imperial Chemical Industries). It is strongly typed with standard data types and control structures. A program consists of a number of independently compiled modules each consisting of an environment description and compilable units called bricks. These are procedure bricks which consist of procedure bodies, data bricks which are like compools and stack bricks which can be created during execution to provide work spaces for tasks. Standard I/O facilities and an error recovery system are not part of the language but are supplied as support tools to increase portability. It has recursive procedures but no nesting of procedure declarations. It has no dynamic arrays. It was designed with the same overall objectives as the proposed DoD common language but reflects the state of the art in 1969.

### Overall Evaluation

RTL/2 appears to be a soundly constructed, relevant language. Its notion of bricks provide a mechanism for scopes and environments more flexible than conventional block structure and its stack bricks allow task creation and real-time facilities to be built up from lower level primitives.

Positive Features

It has strong typing, standard data types and standard control structures.

Its notion of bricks is an interesting and perhaps useful way of blending block structure and with independent modules.

It provides hooks to standard I/O procedures, error recovery procedures and real-time facilities in an operating system.

There are 150-200 real-time systems programmed in RTL/2 of which half are outside ICI. The applications include process control, factory automations, laboratory systems, communications and on-line banking. Many (including all those in ICI) employ operating systems written in RTL/2.

RTL/2 was developed as a result of a requirements evaluation similar to that of TINMAN. The RTL/2 experience of language development and language use might be useful as an input to the design effort. ICI has offered its help, possibly in association with an American Software Company.

Negative Features

It has no extensibility features, precision features for numerical variables or explicit real-time features.

## 3.15 FORTRAN Evaluation

### Evaluators

FORTRAN was evaluated by Intermetrics and IBM.

### Language Description

FORTRAN was developed in the period 1954-58, was essentially the first higher level language, and is still the most widely used language after COBOL. Its subroutine structure and COMMON data blocks was not taken up in block structure languages. However, its lack of data description facilities, control structures, pointer variables and recursive procedures illustrate that there has been some progress in language design since the development of FORTRAN. The FORTRAN evaluation was performed for the 1966 ANSI standard. A revised draft standard for FORTRAN has been prepared and is being considered by ANSI but has not yet been approved. Since it is in principle possible that substantive changes to this draft standard may still be made, it was not considered for purposes of this evaluation.

### Overall Evaluation

Both contractors feel that, in spite of the fact that FORTRAN has shown remarkable resilience and longevity, it is not suitable as a base language because its design does not reflect advances in language design during the last twenty years.

### Positive Features

Simplicity, efficiency, subroutine structure.

### Negative Features

Lack of data types, scope rules, control structures, recursive procedures, pointers, etc.

FORTRAN reflects the language design philosophy of the 1950's and is dominated as a base language by more recently designed programming languages.

## 3.16 COBOL Evaluation

### Evaluators

COBOL was evaluated by Intermetrics and IBM.

### Language Description

COBOL was initially developed in the period 1959-61 as a language for business data processing applications, building on previous languages, such as Univac's Flowmatic and IBM's Commercial Translator. COBOL has had significant additions since its initial development. Its data description facilities and its concept of environment, data and procedure divisions are still important today but the algorithm specification part of the language is behind the state of the art. It is currently the most widely used programming language in the U.S.A. but is not intended for embedded computer applications.

### Overall Evaluation

Both Intermetrics and IBM agree that COBOL is not suitable as a base language for embedded computer applications, both because it was not designed for this class of applications and because it is dominated as a base language by more recently designed languages.

### Positive Features

It seems to be the most widely used higher order language.

It has good I/O and data description facilities and is adequate for specifying simple algorithms.

The distinction between environment, data and procedure divisions is worth while.

### Negative Features

It was designed for business data processing rather than embedded computer applications.

Its English language notation makes programs deceptively readable but does not aid reliability or modifiability.

It has only a two level block structure, inadequate data type facilities, inadequate control structure and procedure definition facilities, and too many special cases and implementation dependencies.

The language definition is too large for its expressive power.

## 3.17 ALGOL 60 Evaluation

### Evaluators

ALGOL 60 was evaluated by SAI.

### Language Description

ALGOL 60 was developed in the period 1957-60 by the IFIP working group WG 2.1. It pioneered the notion of block structure, data type declarations, access scopes, etc., as well as the development of clean programming language definitions. It has gained wider acceptance as a production programming language in Europe than in the U.S.A. It is carefully designed to be implementable in a stack structured run-time environment. It has no explicit input-output facilities and lacks other important features such as record structures, pointer variables, extension facilities, parallel processing, independent (external) procedures, etc. It has influenced the design of almost all subsequently designed block structure languages, including PL/I, PASCAL and ALGOL 68, and is a "grandfather" of more recently designed block structure languages like CS-4, LIS, and HAL/S.

### Overall Evaluation

ALGOL 60 is not directly suitable as a base language, but its influence on the design of the new language will make itself felt since all the recommended base languages are "children" or "grandchildren" of ALGOL 60.

### Positive Features

ALGOL 60 is one of the most important programming languages ever developed and has influenced the design of almost all subsequently designed programming languages. Its definition still serves as a model for current programming language definitions.

Negative Features

Development in programming language methodology since the design of ALGOL 60 make this language inappropriate as a base language. PL/I, PASCAL, and ALGOL 68 have, each in a different way built upon the design philosophy of ALGOL 60 and it is suggested that these languages, or even more recently developed derivative languages be used as a base language.

## 3.18 TACPOL Evaluation

### Evaluators

TACPOL was evaluated by Softech, Intermetrics, and RLG.

### Language Description

TACPOL is a block structured language developed for the Army for a particular application (TACFIRE) on a particular computer (AN/GYK-12). It is a simple, easy to learn, block structured language with fixed point (but not floating point) numeric data, character and bit data, arrays of up to 3 dimensions, records (called groups), tables and overlay facilities. Although all identifiers require data-type, declarations-type checking can be defeated by certain built-in (intrinsic) operations and overlay operations. Procedures can have value parameters, quantity parameters (by reference but without type checking), parameterless procedures and labels. Control structures include if-then-else and iteration, while and case statements. File handling facilities are provided. Exception handling can be performed by passing label parameters (the TACPOL ON statement is deceptive in that it is really a conditional statement rather than a true interrupt). Parallel treatment of input-output and computation may be specified in special cases but there are no general parallel processing facilities. The above description illustrates that TACPOL has a large number of special language features which were included for reasons of efficiency because the inclusion of corresponding cleanly designed general purpose features was not properly understood. The language may be well suited to the particular

application for which it was designed but suffers from restricted expressive power and defects in reliability, portability, and maintainability.

Overall Evaluation

All three language evaluators feel that TACPOL is not suitable as a base language. Its features are a rather small subset of the acceptable features of other recommended languages such as PASCAL (SofTech).

Positive Features

TACPOL is simple and easy to learn.

Negative Features

TACPOL lacks too many of the requirements.

It lacks enumeration types and data definition facilities. There is no floating point data type.

Shortcomings exist in the definition of iterative and conditional control structures, including conditional evaluation and optional ELSE.

Recursion is not well defined. Pointers are not supported. Arrays cannot have dynamic bounds.

Parallel processing is not supported, nor do any time dependent features exist.

Exception handling is inadequate.

There is no facility for describing the object environment or providing an external interface to the object machine.

Certain symbols such as the $=$ symbol are syntactically overloaded.

A considerable amount of defaulting is permitted.

There is inadequate checking for parameters passed by reference. There is no discriminated union facility.

## 3.19 CMS-2 Evaluation

### Evaluators

CMS-2 wes evaluated by RLG and CSC.

### Language Description

CMS-2 is a Navy language developed by the Navy in the period 1966-69. It developed from an earlier language CS-1, and has gone through many updates, each requiring upward compatibility with a previous language. CMS-2 now contains most of the structured programming features common to modern HOLs such as the case statement, WHILE and UNTIL iteration, block structuring, etc. and features considered necessary for embedded military systems, such as machine code insertion, compools, direct procedure linkage, direct access to hardware and machine functions. Some of these features make the language very machine dependent and there are currently several different versions for different computers. The evaluated language is CMS-2-Y for the AN/UYK-7.

### Overall Evaluation

Both RLG and CSC feel that the language is not acceptable as a base language because it lacks too many of the required features, including scope rules, precision specification, pointers, recursion, extensibility, real time facilities, I/O facilities, machine language interface, etc. Although the language appears to be successful in its present applications, it is not portable to new embedded computer applications on new computers.

Positive Features

The language has been successfully used in embedded computer applications and meets, those TINMAN requirements that are met by other languages currently in use for military systems.

Although the language is large, it is not unwieldly or difficult to learn. However, its design may make it larger than necessary for the expressive power it provides.

Negative Features

Many important required features are missing such as scope rules, precision specification, pointers, recursion, extension facilities, parallel processing.

Lack of high level constructs leads to excessive use of machine language. It is estimated that NTDS is coded 30% in machine language.

The source language is defined in a machine dependent manner and therefore even programs written entirely in the HOL part of CMS-2 will not be portable. For example, array assignment, structure accessing, and the shift operator is machine dependent.

Procedure parameters are not local to procedure definition. Procedure linkage mechanism is not conducive to modularity.

Certain language constructs have unexpected and unnecessary constraints. For example, FIND is restricted to one dimentional arrays. Arrays can contain records but records cannot contain arrays.

The scaling and truncation rules of CMS-2 appear to be unnecessarily complex and can cause undesirable errors in numerical computation.

The CMS-2 language has special purpose features which increase translator complexity and may prevent the translators from producing efficient object code.

CMS-2 provides excessive generality at a number of places where it is not needed while imposing unnecessary restrictions at a number of places where generality is appropriate.

CMS-2 has inconsistencies and ambiguities in its definition. Inconsistencies include scaling rules for constants, interpretation of algebraic notation in MEANS and EQUALS statement, value of control variables on loop terminations. Ambiguities include order of expression evaluation.

The definition of CMS-2 is more complex than necessary. This is particularly evident in its discussion of its error prone scaling rules.

CMS-2 does not meet the requirements of simplicity, efficiency, reliability, maintainability or portability. It is an error prone excessively machine dependent language.

## 3.20 SIMULA 67 Evaluation

### Evaluators

SIMULA 67 was evaluated by SofTech.

### Language Description

SIMULA 67 is an extension of ALGOL 60 which contains an important new kind of program module called a class. A SIMULA class just like an ALGOL block consists of a set of declarations followed by a sequence of statements. However, SIMULA separates creation of class data structures from execution of the class body, thereby allowing coroutine control structures and the simulation of objects which have independent data and procedural attributes. The standard SIMULA extension for simulation purposes has interesting simulated time (but no real-time) facilities.

### Overall Evaluation

SIMULA has played an important role in the evolution of concepts of modularity but its module structure does not conform to DoD requirements. It is not recommended as a base language since removal of the class feature from SIMULA would result in a weak ALGOL 60-like language which would be a poor starting point for the design of the common language.

### Positive Features

SIMULA has played an extremely important role in the development of concepts of modularity.

The SIMULA subclass facility is an important and powerful device for providing extensibility in programming languages.

### Negative Features

The requirements of simulation languages appear to differ and in some respects be diametrically opposed to those of

real-time languages, since simulation is specifically concerned with modelling behavior by indirect, rather than direct means. Thus, both efficiency and reliability are of less concern in simulation languages than in real-time languages.

SIMULA has considerably advanced the state-of-the-art but contains transitional as opposed to stable modularity and extensibility features.

The SIMULA class concept supports records only indirectly. In order to meet the DoD requirements, it would be necessary to introduce an additional mechanism for directly realizing records thus resulting in two different language mechanisms for realizing essentially the same language feature.

Although classes are an important and ingeneous language concept, they are two general and high level for the common language. In order to meet the DoD requirements classes would have to be removed and replaced by a set of features (such as records and encapsulated data-types) which directly meet individual DoD requirements. It makes no sense considering SIMULA as a base language if the class concept has to be removed.

## 3.21 JOVIAL J3B Evaluation

### Evaluators

JOVIAL J3B was evaluated by SofTech and CSC.

### Language Description

JOVIAL J3B was developed by Boeing, the Aerospace Corporation and the Air Force in 1972. It is a small language with no dynamic storage allocation or input-output. It has some nice features, such as local, global, systems and COMPOOL scopes, evaluation of constants at compile time and conditional compilation for if-then-else statements whose condition can be evaluated at compile time. However, it does not have precision specifications, extension facilities, parallel processing, exception handling or real time facilities.

### Overall Evaluation

JOVIAL J3B is inappropriate as a base language because it lacks too many of the required features and in addition has numerous arbitrary restrictions and inconsistencies (SofTech evaluation) which would have to be removed or resolved before J3B could serve as a suitable basis.

### Positive Features

JOVIAL J3B has a number of nice language features.

### Negative Features

Too many of the substantive DoD requirements are not in the language. It does not have fixed point numbers, precision specifications for variables, enumeration types, type definition and extension facilities, recursive procedures, arrays with dynamic bounds, parallel processing, exception

handling and real-time facilities.

It has some inconsistencies in areas such a parameter passing so that deletions from the small existing language would be required before extending the language to meet the TINMAN requirements.

## 3.22 JOVIAL J73 (level 1) Evaluation

### Evaluators

JOVIAL J73 (level 1) was evaluated by Intermetrics and CSC.

### Language Description

JOVIAL J73 was designed by an Air Force Committee in the period 1969-73. It has integer, floating point, character and bit data types and aggregate data types called tables and blocks. Although all identifiers must be introduced by type declarations, the language is not strongly typed because overlays permit variables of one type to be used as if they had another type and because there is no type checking in the use of table data and in passing procedure parameters. JOVIAL J73 lacks I/O facilities, does not have enumeration types, does not have mode definition facilities and does not have parallel processing exception handling or real-time facilities.

### Overall Evaluation

JOVIAL J73 fails to meet an exceptionally large number of the DoD requirements. It is not strongly typed, has no array and record types, fixed point variables, enumeration types, extension facilities, recursive procedures, parallel processing facilities, exception handling facilities, real-time facilities, I/O facilities. Its defining document differs from any of the current implementations. It is not recommended as a base language.

### Positive Features

JOVIAL J73 is designed to be simply compilable and to generate efficient code.

### Negative Features

JOVIAL J73 is poor with regard to reliability, transportability, and machine independence.

It lacks strong typing, array and record types, fixed point variables, enumeration types, extension facilities, parallel processing, real time and exception handling features, and I/O facilities.

There are problems concerning the definition of J73. Currently, there are several different versions of J73 including a draft standard which differs from any implemented language.

The parameter passing mechanism is inadequate both because it does not enforce type checking and because it imposes arbitrary restrictions on passing table and blocks.

JOVIAL J73 lacks a machine independent interface to the operating system and has no facility for machine code insertions.

It does not specify the order of expressing evaluation and therefore allows optimizations which can change the effect of the program.

## 3.23 CORAL 66 Evaluation

### Evaluators

CORAL 66 was evaluated by CSC and RLG.

### Language Description

CORAL 66 was designed in 1966 by the British Royal Radar Establishment (RRE). It was adopted in the 1970's as an inter-service standard for military programming and has been widely used in the British control on automation industry. It has block structured scope, as well as independently compilable segments, global COMMON data, and overlay declarations. Its data types include floating, fixed, and integer (but not Boolean or character), arrays, packed data and table declarations (in place of records). Recursive procedures must be explicitly designated. There are no parallel processing or real time facilities. Error conditions may be handled by label parameters. Code statements which allow escape into machine language are part of the language.

### Overall Evaluation

CORAL 66 is too small and early a language to be recommended as a base language. However, its clean ALGOL 60-like design is impressive and its considerable use in Britain for military and industrial applications must be respected.

### Positive Features

A clean design which incorporates both block structure and independently compilable program and data segments.

Considerable use in both military and industrial applications.

Negative Features

It was designed too early to incorporate notions such as record structures, reference variables, enumeration types and other ALGOL 68 and PASCAL language features.

It does not contain Boolean variables, precision specifications, mode definition.

It does not contain parallel processing, real-time features or clean exception handling.

# 4. Discussion, Recommendations, and Conclusions

## 4.1 Discussion

The Statement-of-Work for the language evaluations called for an analysis of how each of the languages matched each of the requirements found in the TINMAN. As part of the attempt to consolidate all the evaluation inputs, a matrix was constructed of TINMAN requirements vs particular languages. The matrix was filled in by mapping contractor's individual scoring systems into a unified joint score followed by an averaging where more than one evaluation was done for one language. The information provided by this matrix was of little use for the following reasons:

* Only the specific (as opposed to general) language requirements were considered in constructing these numbers (i.e., TINMAN sections A-J).

* There was no attempt to weigh the relative importance of each requirement relative to each of the other requirements. Each Requirement was considered equal.

* If a language did not satisfy some requirement, the difficulty or ease of making the required change was not distinguished in the matrix numbers.

* In certain cases, if a language did not satisfy some requirement it would affect many other requirements as well. This tended to magnify unduely the overall effect of the missing requirement on the score.

* Large languages with lots of features tended to score higher than smaller languages.

* The scoring did not reflect the fact that even though a language might satisfy a particular requirement (perhaps only partially), it might be the case that a better way of satisfying the requirement entailing a significant amount of work might make the language much better.

* Different evaluators, even when they basically agreed on some point, tended to score differently. SofTech was consistently more severe than say IBM which was rather lenient.

* Some of the languages had only one evaluation while others had up to four evaluations.

* Some evaluations were in a form where the information could not be included in this matrix.

The most useful information to the coordinating committee was obtained by detailed discussions with evaluators after their studies had essentially been completed.

### 4.2 Recommendations

The first question that had to be answered by the language evaluations was: "Does a language already exist that satisfies the DoD set of language requirements as set forth in the TINMAN, so well that it can be adopted as the Common Language with little or no changes required?" With unanimous agreement, none of the evaluators found that such a language now exists.

The consensus of the evaluations was that it would be possible to produce a single language meeting essentially all the DoD language requirements, hence the next efforts should be directed toward the goal of producing a single language.

Each feature or capability mentioned in the requirements document can essentially be found in some existing language, hence some minimal collection of languages can be found that collectively contains all these features and capabilities. However, there are important embedded computer system applications that could make good use of all the major requirements. In fact, most of the requirements would be useful in any embedded computer application. It is clearly possible to design a language by brute force containing all the technical features and capabilities of the requirements. Problems arise, however, when one adds requirements such as simplicity, uniformity, reliability, design integrity, implementation efficiency, etc. Producing a language that satisfies all of the requirements is not a straight-forward task. The consensus was, nonetheless, that such a language can be produced and work toward the development of a single language should proceed.

Some evaluators did feel that some of the requirements will have to be modified. Many of these have already been incorporated in the new IRONMAN document. Others, concerned for example with the precise degree of trade-off between potentially conflicting requirements (such as simplicity and generality) can be determined only after a substantial amount of further work on the design of the language.

Just about all the evaluators felt that work toward the production of a language satisfying the requirements should start from some carefully chosen base language. Working from such a base language would reduce the amount of work requiree and would reduce the opportunities for making errors. There was no consensus as to which base language to use, but every evaluator indicated that some of the languages considered were more suitable for this role than others. Each felt in fact that it would make a substantial difference depending on the base language chosen. There was remarkable agreement concerning the unsuitability of certain languages to serve as base languages.

Even though starting with some base was recommended, all felt that a design team must have the freedom to make any changes they feel are warranted. For example, upward compatibility with the base language should <u>not</u> be required. The specification for a design effort must be carefully written so as not to place any artificial or undesirable restrictions on the design team.

The role of a base language in a design effort is difficult to define, and it appears that any attempt to define it will not be worth the effort. A well-qualified design team working from a good base language will know how to proceed without such a definition.

With a unanimous degree of agreement, the following languages were found by the evaluations to be inappropriate to serve as base languages: FORTRAN, COBOL, TACPOL, CMS-II, JOVIAL J73, JOVIAL J3B, SIMULA 67, ALGOL 60, and CORAL 66.

This should not be interprested as a statement concerning the technical merits of these languages, it is only a statement by the evaluators that better languages exist for the role of base languages for a design effort toward the Common Language. Some of the languages on this list of rejected languages are among the most widely used languages. The detailed reasons for their unsuitability can be found in the evaluation documents (See Sec.5). We have made an attempt to bring this information together in Section 3.

The languages that remain are: PASCAL, PL/I, ALGOL 68, LIS, EUCLID, CS-4, PDL/2, RTL/2, LTR, PEARL, SPL-1, HAL/S, MORAL, and ECL.

The following recommendation appears to be consistent with almost all the evaluations: Proposals should be solicited from appropriate language designers for modification efforts using any of the languages PASCAL, PL/I, or ALGOL 68 as base languages.

Some of the languages, among those listed above that were not found inappropriate to serve as base language, can play the following roles. The languages PL/I, PASCAL, and ALGOL 68 each represent a different synthesis of a large amount of design experience and each constitutes a nucleus of a family of derived languages. The languages EUCLID, PDL/2, LIS, LTR, and CS-4 are in the PASCAL family and to the extent that a design team feels these experiences are useful, they can be used as appropriate. The language EUCLID, although somewhat of a research language (i.e., one still under development)

is very carefully designed to support the writing of verifiable well-structured programs and contains many interesting relevant properties that should be helpful to a PASCAL modification effort. CS-4 has also not been completely designed and tested, but the language is relevant and should be considered by such a design effort. The language LIS contains a very interesting treatment of the requirement for separate compilation among other things and should be useful to design teams no matter what base is used. The language LTR is amodern real-time language standard used in France for a wide range of applications.

A similar case can be made for the languages HAL/S and PEARL relative to PL/I, and RTL/2 relative to ALGOL 68. Again, the designers should have the option to use this information to the extent they deem appropriate.

After the designers have developed their ideas to an appropriate degree, an evaluation of these three modification efforts should be made. This should determine which should be further supported to completion.

## 4.3 Conclusions

The HOLWG language evaluation study is an interesting and significant application of current techniques of software engineering to the problem of programming language design. A programming language may be regarded as having a life cycle with a requirements analysis and design phase,

an implementation and debugging phase and an operations and maintenance phase. Programming languages just as large application programs have in the past been designed with insufficient emphasis on requirements analysis. The present study has emphasized requirements analysis and in so doing has considerably increased our understanding of the nature of requirements specification for programming languages. Thus, this study should be helpful not only in the design of the DoD common language but also in the design of other programming languages.

A great deal of work on detailed design and implementation of the common language still remains to be done. However, the language evaluation study has resulted in a clear recommendation to proceed with the design of the common language by modification of a carefully chosen base language. Moreover the language evaluations study has allowed us to systematically evaluate the suitability and relevance of a large number of currently important programmed languages. The information concerning programming languages accumulated as a result of the language evaluation effort should prove useful not only in choosing a base language, but also in determining how a chosen language should be modified to meet the DoD requirements.

## 5. A GUIDE TO THE SUPPORTING DOCUMENTS

### 5.1 Language Requirements

D.A. Fisher, "A Common Programming Language for the Department of Defense -- Background and Technical Requirements," Institute for Defense Analyses Paper P-1191, June 1976.

"Department of Defense Requirements for High Order Computer Programming Languages" - "TINMAN" Defense Advanced Research Projects Agency, publication (known as "TINMAN"), June 1976.

"Department of Defense Requirements for High Order Computer Programming Languages" - "IRONMAN" January 1977.

### 5.2 Documents used as Basis for Evaluation

#### 5.2.1 FORTRAN

"American National Standard (ANS) FORTRAN," ANSI X3.3-1966, American National Standards Institute, Inc., New York, March 196

"Draft Proposed American National Standard FORTRAN," BSR X-3.3 X3J3/76, American National Standards Committee X3J3, March 1976. (also published in SIGPLAN Notices, Vol. II, No.3, March 1976).

#### 5.2.2 COBOL

"American National Standard Programming Language COBOL, "ANSI X3.23-1974, American National Standards Institute, Inc., New York, 1974 (less the Report Writer Module).

5.2.3 PL/I

"Draft Proposed American National Standard Programming Language PL/I; Basis /1-12" Document BSR X353, Feb. 1975.

"Errata for BSR X353" Document X3J1/399, Jan. 1976.

5.2.4 TACPOL

"TACPOL Reference Manual" USA CSCS-TF-4-1, Data Systems Division, Litton Systems, Inc. Jan. 1972.

"CPCEI SPECIFICATION FOR COMPILER/ASSEMBLER FOR FIRE DIRECTION SYSTEM,"
Spec. No. EL_CG-00043082C, Doc. No. 595909, Vol.1 of 2, Appendix 10, Data Systems Division, Litton Systems, Inc. April 1971 (Formal Definition of TACPOL).

5.2.5 HAL/S

"HAL/S Language Specification," Intermetrics, Inc., Cambridge, Mass., April 1973.

5.2.6 CMS-2

"Users Reference Manual for Compiler Monitor System-2 (CMS-2) for use with the AN/UYK-7 Computer" M-5035, Vol.1 and 2. FCDSSA - San Diego, Cal. 15 Aug. 1975.

"CMS-2Y Programmer's Reference Manual (Preliminary Version)" M50- FCDSSA - San Diego, Cal. 1 Oct. 1976.

5.2.7 CS-4

"CS-4 Language Reference Manual and Operating System Interface." Advanced Software Technology Division, NELC, San Diego, Cal. Oct. 1975.

"CS-4 Primer, Vol.I, Basic Features." Advanced Software Technology Division, NELC, San Diego, California, Feb. 1975.

5.2.8 JOVIAL/J3B

"JOVIAL/J3B Extension 2 Language Specification" Softech, Inc. Waltham, Mass., Document 2044-4.2, July 1975.

5.2.9 JOVIAL /J73

"JOVIAL J73/1 Specification" RADC Document, July 1976.

5.2.10 ALGOL-60

P. Naur (Editor), "The Revised Report on the Algorithmic Language ALGOL-60." Communications ACM, Jan. 1963.

R. Baumann, M. Feliciano, F.L. Bauer, and K. Samuelson, Introduction to ALGOL. Prentice-Hall, Englewood Cliffs, N.J. 196

R.M. DeMorgan, I.D. Hall and B.A. Wichman, "A Supplement to the ALGOL 60 Revised Report," The Computer Journal, Vol. 19, No. 3, 1976.

5.2.11 CORAL 66

"Inter-Establishment Committee on Computer Applications Official Definition of CORAL 66" Ministry of Defense Her Majesty's Stationary Office London, 1970.

5.2.12 ALGOL-63

A.Van Wijngaarden, et.al. (eds) "Revised Report on the Algorithmic Language ALGOL 63. Springer-Verlog, 1976.

A. Van Wijngaarden, et.al., "Revised Report on the Algorithmic Language ALGOL 63" Acta Informatica, Vol. 5, pages 1-236, 1975.

A.S. Tanenbaum, "A Tutorial on ALGOL 68." Computing Surveys, Vol. 8, No. 2, June 1976.

5.2.13 SIMULA 67

O.J. Dahl, B. Myhrhaug, and K. Nygaard, "SIMULA 67, Common Base Language," Norwegian Computing Center Publication No. S-22, 1970.

P. Naur (Ed.), "Revised Report on the Algorithmic Language ALGOL 60," Comm. ACM, January 1963, Vol.6, No.1, pp 1-17.

A. Birtwistle, O. J. Dahl, B. Myhrhaug, and K. Nygaard, "SIMULA BEGIN," Auerbach Publishers Inc., Philadelphia, 1973.

J. Palme, "SIMULA as a Tool For Extensible Program Products, "ACM SIGPLAN Notices," Feb. 1974, Vol.9,No.2, pp.24-40.

NCC System Programming Group, "The Structure of the NCC SIMULA, Compilers and Bench Mark Comparisons with other Major Languages, "Norwegian Computing Center Publication" No. S-43, 1972.

5.2.14 PASCAL

K. Jensen and N.Wirth, "PASCAL User Manual and Report," Springer-Verlog, 1974.

N. Wirth, "An Assessment of the Programming Language PASCAL," IEEE Trans. on Software Engineering, 1, 192-198. (1975).

N. Wirth, "The Design of a PASCAL Compiler," Software - Practice and Experience, 1, 309-333, (1971).

5.2.15 LIS

J.D. Ichbiah, J.P. Rissen, and J.C. Heliary, "The Two-Level Approach to Data Independent Programming in the LIS System Implementation Language." Compagnie Inter. Pour L'Informatique, Louveciennes, France, June 1975.

"The System Implementation Language LIS," CII. Document No. 4549 EL/EN, Jan. 1976.

J.D. Ichbiah, J.C. Heliard, "Plexes in LIS, 1975.

J.D. Ichbiah and P. Cousot, "Visibility and Separate Compilations," 1975.

5.2.16 RTL/2

"RTL/2 Language Specification for RTL/2 Reference: 1 Version: 2", Imperial Chemical Industries Limited, 1964

5.2.17 EUCLID

B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, G.J. Popek, "Report on the Programming Language EUCLID"

5.2.18 MORAL

"A Description of the Programming Language MORAL" Reference No: 22/1132 Software Sciences Limited, Feb. 1976.

"Mascot - A Modular Approach to Software Construction Operation and Test" Technical Note No. 778, Royal Radar Establishment, Procurement Executive, Ministry of Defense, Malvern, Worcs.

5.2.19 PEARL

"PEARL - Subset for Avionic Applications Language Description" ESG Elektronik-System-Gesellschaft M6H, June 197[illegible].

5.2.20 SPL/I

"Reference Manual for SPL/I" Intermetrics, Inc.

5.2.21 PDL/2

"Process Design Methodology Design Specification, Vol.1, Process Design Language" Sept. 1976, Texas Instr. Inc.

5.2.22 LTR

"LTR Reference Manual" (French) Centre de Programmation de la Marine, Paris December 1976.

"LTR Programmers Manual: Centre de Programmation de la Marine, Paris December 1976.

5.2.23 ECL

"Reference Manual for ECL" Harvard University, 1974.

5.3 Documents produced by Language Evaluators

5.3.1 "HOL Evaluation Project Interim Technical Report," Science Applications, Inc., Software Technology Center, 211 Sutter Street, San Francisco, California 94103.

5.3.2 "Evaluation of CORAL 66" AD-A039636

"Evaluation of PASCAL"

"Evaluation of CS-4"

"Evaluation of TACPOL"

"Evaluation of CMS-2

RLG Associates, Inc. 11250 Roger Bacon Dr., Reston, Va. 22090

AD A037637

5.3.3 "Evaluation of ALGOL 68, JOVIAL J3B, PASCAL, SIMULA 67 and TACPOL vs. TINMAN Requirements for a Common High Order Programming Language," Oct. 1976. Softech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154

5.3.4 "Draft Candidate Language Evaluation and Recommendation Report" I.B.M. AD A037638

5.3.5 B.L. ~~Maps~~ Marks, "Some Characteristics for a Common DoD Programming Language and their Relation to PL/I" IBM, United Kingdom Laboratories Limited, Harsley Park, Winchester.

5.3.6 "Draft Version - Candidate Languages Evaluation and Recommendation Report," Dec. 1976, Intermetrics, Inc., 701 Concord Ave., Cambridge, Mass. 02138. AD A037639

5.3.7 "Candidate Language Evaluation and Recommendation Report" Computer Sciences Corp., 6565 Arlington Boulevard, Falls Church, Va. 22046 AD A037640

5.3.8 L.Campbell, "Comparative Evaluation TINMAN VS. FORTRAN" BRL, APG, Aberdeen, Md. August 1976.

5.3.9 A. Demers, "Evaluation of LIS, LTR, RTL/2" December 1976.

5.3.10 S. Squires, "Draft Evaluation of ECL with Respect to "TINMAN" Dec. 1976.

5.3.11 P.Parayre, "LTR Evaluation towards "TINMAN" Requirements" (From Centre de Programmation de la Marine, Paris).

5.3.12 J.G.P. Barnes, "An Informal Evaluation of RTL/2 against TINMAN" Oct. 1976. (From Imperial Chemical Industries, Ltd.)

5.3.13 "An Assessment of the Programming Language MORAL against TINMAN" Oct. 1976, Software Sciences Ltd., Abbey House, 282-292 Farnborough Rd., Farnborough Hampshire, England.

5.3.14 "Language Evaluation of PDL/2" Texas Instruments Dec. 1976.

5.3.15 J. Williams, "An Evaluation of PEARL" Cornell Univ., Jan.1977 AD-A037641

5.3.16 D. Morris et.al.,"DoD-1 Language Evaluation Matrix" Jan.1977.

5.3.17 J. D. Ichbiah, "A Comparative Evaluation of the LIS Language with Respect to the TINMAN Requirements" Oct. 1976

5.3.18 Thomas Martin, "A Comparative Evaluation of PEARL, the Process and Experiment Automation Realtime Language, against the DOD-TINMAN-Requirements for High Order Computer Programming Languages" June 1976

5.3.19 "An Assessment of the Programming Language CORAL 66 Against The U.S. Department of Defense TINMAN Requirements" Nov. 1976, Software Sciences Ltd., Abbey House, 282-292 Farnborough Rd., Farnborough, Hampshire, England

5.1

# DEPARTMENT OF DEFENSE

# REQUIREMENTS FOR HIGH ORDER

# COMPUTER PROGRAMMING LANGUAGES

"IRONMAN"

14 January 1977

## THE TECHNICAL REQUIREMENTS

The technical requirements for a common DoD high order programming language given here are a synthesis of the requirements submitted by the Military Departments. They specify a set of language characteristics that are appropriate for embedded computer applications (i.e., command and control, communications, avionics, shipboard, test equipment, software development and maintenance, and support applications).

The changes that produced this revision reflect the many comments on previous versions received from the Services, military contractors, the research community, and other organizations during 1976. This revision does not alter the basic intent or substance of the December 1975 (i.e., "TINMAN") version of the requirements. It does incorporate changes to improve clarity, to correct errors, and to ensure feasibility.

The revised requirements are hierarchically organized with an outline similar to that expected in a language defining document. Section 1 gives the general design criteria. These provide the major goals that influenced the selection of specific requirements and provide a basis for language design decisions that are not otherwise dealt with in the requirements. Section 2 through 12 give more specific technical requirements on the language and its translators. The requirements call for the inclusion of features to satisfy specific needs in the design, implementation, and maintenance of military software, specify many general and specific characteristics desired for the language, and call for the exclusion of certain undesirable characteristics. Section 13 gives some of the intentions and expectations for development, control, and use of the language. The intended use and environment for the language has strongly influenced the requirements; understanding those intentions should aid in achieving the requirements.

A precise and consistent use of terms has been attempted throughout the requirements. Potentially ambiguous terms have been defined in the text. Care has been taken to distinguish between requirements, given as text, and comments about the requirements, given as bracketed notes. Previously duplicative requirements have been given just once. Potentially conflicting implications of requirements have been clarified.

The following terms have been used throughout the text to indicate where and to what degree individual requirements apply:

| | |
|---|---|
| shall | indicates a requirement placed on the language or translator |
| should | indicates a desired goal but one for which there is no objective test |
| shall attempt | indicates a desired goal but one that may not be achievable given the current state of the art, or may be in conflict with other more important requirements |
| shall require | indicates a requirement that is to be placed on the user by the language and its translators |
| shall permit | indicates a requirement placed on the language to provide an option to the user |
| must | same meaning as shall require but takes user as subject |
| may | same meaning as shall permit but takes user as subject |
| will | indicates a consequence that is expected to follow or indicates an intention of the DoD; it does not in any case by itself constrain the design of the language. |

Some of the above terms are also used informally in the bracketed notes. Language is used in the singular to refer to the minimum number of languages necessary to satisfy the needs of DoD applications.

## 1. General Design Criteria

**1A. Generality.** The language shall provide generality only to the extent necessary to satisfy the needs of embedded computer applications. Such applications require real time control, self diagnostics, input-output to nonstandard peripheral devices, parallel processing, numeric computation, and file processing. The language shall not contain features that are unnecessary to satisfy the requirements.

**1B. Reliability.** The language should aid the design and development of reliable programs. The language shall be designed to avoid error prone features and to maximize automatic detection of programming errors. The language shall require some redundant, but not duplicative, specifications in programs. Translators shall produce explanatory diagnostic and warning messages, but shall not attempt to correct programming errors.

**1C. Maintainability.** The language should promote ease of program maintenance. It should emphasize program readability over writability. That is, it should emphasize the clarity, understandability, and modifiability of programs over programming ease. The language should encourage user documentation of programs. It shall require explicit specification of programmer decisions and shall provide defaults only for instances where the default is stated in the language definition, is always meaningful, reflects common usage, and can be explicitly overridden.

**1D. Efficiency.** The language design should aid the production of efficient object programs. Constructs that have exceptionally expensive or exceptionally inexpensive implementations should be easily recognizable by translators and by users. Users shall be able to specify the time space trade offs in a program. Where possible, features should be chosen to have a simple and efficient implementation in many object machines, to avoid execution costs for available generality where it is not needed, to maximize the number of safe optimizations available to translators, and to ensure that unused and constant portions of programs will not add to execution costs. Execution time support packages of the language shall not be included in object code unless they are called.

**1E. Simplicity.** The language should not contain unnecessary complexity. It should have a consistent semantic structure that minimizes the number of underlying concepts. It should be as small as possible consistent with the needs of the intended applications. It should have few special cases and should be composed from features that are individually simple in their semantics. The language should have uniform syntactic conventions and should not provide several notations for the same concept.

**1F. Implementability.** The language shall be composed from features that are understood and can be implemented. The semantics of each feature should be sufficiently well specified and understandable that it will be possible to predict its interaction with other features. To the extent that it does not interfere with other requirements, the language shall facilitate the production of translators that are easy to implement and are efficient during translation. There shall be no language restrictions that are not enforceable by translators.

**1G. Machine Independence.** The language shall strive for machine independence. It shall not dictate the characteristics of object machines or operating systems. The design of the language shall attempt to avoid features whose semantics depend on characteristics of the object machine or of the object machine operating system. There shall be a facility for specifying those portions of programs that are dependent on the object machine configuration and for conditionally compiling programs depending on the actual configuration.

**1H. Formal Definition.** To the extent that a formal definition assists in achieving the above goals, the language shall be formally defined. [Note that formal definitions are of most value during language design; and that the same method may not be appropriate for defining all aspects of a language.]

## 2. General Syntax

**2A. Character Set.** Every construct of the language shall have a representation that uses only the 64 character subset of ASCII:

```
 !"#$%&' ()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
```

**2B. Grammar.** The language should have a simple, uniform, and easily parsed grammar and lexical structure. The language shall have free form syntax and should use familiar notations where such use does not conflict with other goals.

**2C. Syntactic Extensions.** The user shall not be able to modify the source language syntax. *In particular the user shall not be able to modify or introduce* new precedence rules or to define new syntactic forms.

**2D. Other Syntactic Issues.** Multiple occurrences of a language defined symbol appearing in the same context shall not have essentially different meanings. The language shall not permit unmatched parentheses of any kind (e.g., begin and end must be paired one for one). Source program line boundaries shall be treated like spaces. All key word forms that contain declarations or statements shall be bracketed (i.e., shall have a closing as well as an opening key word).

### 2.1. Identifiers

**2E. Mnemonic Identifiers.** Mnemonically significant identifiers shall be allowed. There shall be a break character for use within identifiers. The language and its translators shall not permit identifiers or reserved words to be abbreviated.

**2F. Reserved Words.** The only reserved words shall be those that introduce special syntactic forms or that are otherwise used as delimiters. Words that can be used in place of identifiers shall not be reserved (e.g., names of built-in or predefined functions, types, constants, and the like shall not be reserved). All reserved words shall be listed in the language definition.

## 2.2. Literals

**2G. Numeric Literals.** There shall be built-in numeric literals. Numeric literals shall have the same values in programs as in data.

**2H. String Literals.** There shall be built-in string literals. String literals shall be interpreted as fixed length one-dimensional character arrays. Literal strings shall not be allowed to cross line boundaries of the source program.

## 2.3. Comments

**2I. Comments.** The language shall allow comments to be embedded within program text (e.g., a comment bracketed by special left and right bracket symbols) and shall allow stand alone comments (e.g., a comment introduced by a special symbol at the beginning of each line). Bracket symbols shall consist of no more than two characters each. The language shall not permit comments to automatically cross line boundaries.

## 3. Types

**3A. Strong Typing.** The language shall be strongly typed. That is, the type or mode of each variable, array and record component, expression, function, and parameter shall be determinable at translation time.

**3B. Implicit Type Conversions.** There shall be no implicit conversions between types.

**3C. Type Definitions.** It shall be possible to define new data types in programs. Type definitions shall be processed entirely at translation time. The scope of a type definition shall be determinable at translation time. No restriction shall be imposed on defined types unless it is imposed on all types.

### 3.1. Numeric Types

**3-1A. Numeric Values.** The language shall provide types for integer, fixed point, and floating point numbers. Numeric operations and assignment that would cause the most significant digits of numeric values to be truncated (e.g., when overflow occurs) shall constitute an exception situation.

**3-1B. Numeric Operations.** There shall be built-in operations (i.e., functions) for conversion between numeric types. There shall be built-in operations for addition, subtraction, multiplication, division with floating point result, and negation for all numeric types. There shall be built-in equality (i.e., equal and unequal) and ordering operations (i.e., less than, greater than, less or equal, and greater or equal) between elements of each numeric type. Numeric values shall be equal if and only if they represent exactly the same abstract value. The semantics of all built-in numeric operations shall be included in the language definition. [Note that there might also be standard library definitions for numeric functions such as exponentiation.]

### 3.1.1. Floating Point Type

**3-1C. Floating Point Precision.** The precision of each floating point variable and expression shall be specifiable in programs and shall be determinable at translation time. Precision specifications shall be required for each floating point variable. Precision shall be interpreted as the minimum precision to be implemented in the object machine. Floating point results shall be implicitly rounded (or on some machines truncated) to the implemented precision. Explicit conversion operations shall not be required between floating point precisions.

**3-1D. Floating Point Implementation.** A floating point computation may be implemented using the actual precision, radix, and exponent range available in the object machine hardware. There shall be built-in operations to access the actual precision, radix, and exponent range with which floating point variables and expressions are implemented.

### 3.1.2. Integer and Fixed Point Types

**3-1E. Integer and Fixed Point Numbers.** Integer and fixed point numbers shall be treated as exact numeric values. There shall be no implicit truncation or rounding in integer and fixed point computations.

**3-1F. Integer and Fixed Point Variables.** The range of each integer and fixed point variable must be specified in programs and determinable at translation time. Such specifications shall be interpreted as the minimum range to be implemented. Explicit conversion operations shall not be required between numeric ranges.

**3-1G. Fixed Point Scale.** The scale or step size (i.e., the minimal representable difference between values) of each fixed point variable must be specified in programs and be determinable at translation time.

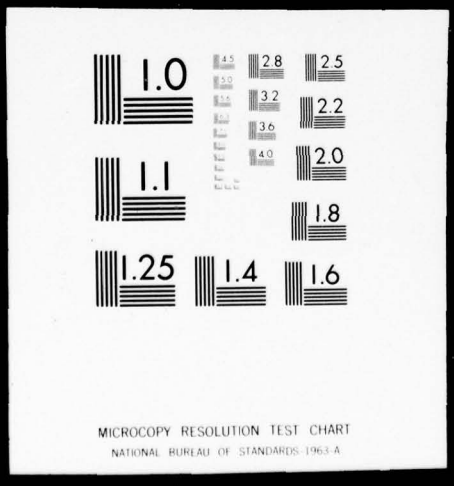
**3-1H. Integer and Fixed Point Operations.** There shall be built-in operations for integer and fixed point division with remainder and for conversion between fixed point scale factors. The language shall require explicit scale conversion operations whenever the scale of a value must be changed to properly perform some operation (e.g., assignment, comparison, or parameter passing).

AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

2 OF 28
ADA037634

2 OF 28
ADA037634



## 3.2. Enumeration Types

**3-2A. Enumeration Type Definitions.** There shall be types that are definable in programs by enumeration of their elements. The elements of an enumeration type may be identifiers or character literals. Literal identifiers shall be syntactically distinguishable from other identifiers. Equality and inequality shall be automatically defined between elements of each enumeration type.

**3-2B. Ordered Enumeration Types.** Ordered enumeration types must be so marked in their definitions. The four ordering operations shall be automatically defined between elements of each ordered type defined by enumeration. A variable of an ordered enumeration type may be restricted to a contiguous subsequence of the enumeration.

### 3.2.1. Boolean Type

**3-2C. Boolean Type.** There shall be a predefined unordered enumeration type for Boolean values. The Boolean type shall have operations for conjunction, inclusive disjunction, and negation.

### 3.2.2. Character Types

**3-2D. Character Types.** Character sets shall be definable as enumeration types. Character types may contain both printable and control characters. Definitions for ASCII and other widely used character sets shall be available in a standard library.

## 3.3. Composite Types

**3-3A. Composite Type Definitions.** It shall be possible to define types that are Cartesian products of other types. Composite types shall include arrays (i.e., composite data with indexible components of homogeneous types) and records (i.e., composite data with labeled components of heterogeneous type).

**3-3B. Component Specifications.** For elements of composite types, the type of each component (i.e., field) must be explicitly specified in programs and determinable at translation time. Components may be of any type (including array and record types). Range, precision and scale specifications shall be required for each component of appropriate numeric types.

**3-3C. Operations on Composite Types.** A value accessing operation shall be automatically defined for each component of composite data elements. Assignment shall be automatically defined for components that have alterable values. A constructor operation (i.e., an operation that constructs an element of a type from its constituent parts) shall be automatically defined for each composite type. An assignable component may be used anywhere in a program that a variable of the component's type is permitted.

### 3.3.1. Arrays

**3-3D. Array Specifications.** The number of dimensions for each array must be specified in programs and shall be determinable at translation time. The range of subscript values for each dimension must be specified in programs and shall be determinable by the time of array allocation. The range of subscript values shall be restricted to a contiguous sequence of integers or to a contiguous sequence from an enumeration type. [Note that translators may be able to produce more efficient object programs where subscript ranges are determinable at translation time.]

**3-3E. Operations on Subarrays.** There shall be built-in operations for value access, assignment, and catenation of contiguous sections of one-dimensional arrays of the same component type.

### 3.3.2. Records

**3-3F. Operations on Records.** Assignment shall be permitted between records with corresponding components of identical name and type.

**3-3G. Nonassignable Record Components.** It shall be possible to specify record components (including tag fields) for which assignment shall not be permitted. These components shall include those defined as constants and those defined as expressions. [Note that such components need not take data storage space.]

**3-3H. Variant Types.** It shall be possible to define types with alternative record structures (i.e., variants). The structure of each variant shall be determinable at translation time. Each variant must have a tag field (i.e., component that can be used to discriminate among the variants during execution). The value of a variant may be used anywhere a value of the variant type is permitted.

### 3.3.3. Types Requiring Dynamic Allocation

**3-3I. Definitions of Dynamic Types.** It shall be possible to define types whose elements are dynamically allocated. Elements of such types may have components of their own type and may have substructure that can be altered during execution. Such types shall be distinguishable from other composite types in their definitions. [Note that such types require pointers and heap storage in their implementations. They are intended primarily for the support portions of embedded computer software.]

**3-3J. Constructor Operations.** Each execution of the constructor operation for a dynamically allocated type shall create a distinct element of the type. Such elements shall remain allocated as long as there is an access path to them.

## 3.4. Set Types

**3-4A. Set Type Definitions.** It shall be possible to define types as power sets of enumeration types. [Note that the elements of such types are sets and can be implemented as bit strings.]

**3-4B. Operations on Sets.** Membership and constructor operations shall be defined automatically for each type defined as a power set. Intersection, union, symmetric difference, equality, and inequality shall be automatically defined between elements of each set type. [Note that intersection, union, and symmetric difference can be implemented as bit by bit operations for conjunction, inclusive disjunction, and exclusive disjunction, respectively.]

## 3.5. Encapsulated Types

**3-5A. Encapsulated Definitions.** It shall be possible to encapsulate definitions. Encapsulations may contain definitions of the data elements comprising a type and of operations.

**3-5B. Effect of Encapsulation.** The effect of encapsulation shall be to inhibit external access to implementation properties of the definition. In particular declarations made within an encapsulation shall not automatically be accessible outside the encapsulation. Data elements defined in an encapsulation shall not automatically inherit the operations of the types with which they are represented.

**3-5C. Own Variables.** It shall be possible within encapsulations to declare variables that are accessible only within the encapsulation but remain allocated throughout the scope in which the encapsulation is declared. Such variables shall retain their values between entries to the encapsulation. It shall be possible to initialize such variables at the time of their apparent allocation.

**3-5D. Operations Between Types.** It shall be possible to define operations, like type conversion, that require access to local properties of more than one encapsulated definition. [Note that this capability violates the purpose of encapsulation and thus its use should be avoided wherever possible.]

## 4. Expressions

**4A. Form of Expressions.** The form (i.e., context free syntax) of expressions shall not depend on the types of their operands or on whether the types of the operands are built into the language.

**4B. Type of Expressions.** The language shall require that the type of each expression be determinable at translation time. It shall be possible to specify the type of an expression explicitly. [Note that the latter requirement provides a way to resolve ambiguities in the types of literals and to assert the type of results; it does not provide a mechanism for type conversion.]

**4C. Side Effects.** The language should permit few side effects in expressions. In particular, during expression evaluation assignment shall not be allowed to any variable that is accessible in the scope of the expression.

**4D. Allowed Usage.** Expressions of a given type shall be allowed wherever both constants and variables of the type are allowed.

**4E. Constant Valued Expressions.** Constant valued expressions (i.e., expressions whose values are determinable at translation time) shall be allowed wherever constants of the type are allowed. Such expressions shall be evaluated before execution time.

**4F. Operator Precedence Levels.** The precedence levels (i.e., binding strengths) of all infix operators shall be specified in the language definition, shall not be alterable by the user, shall be few in number, (e.g., three or four), and shall not depend on the types of the operands. [Note that there might be built-in operator symbols whose meaning is entirely specified by the user.]

**4G. Effect of Parentheses.** Explicit parentheses shall dictate the association of operands with operators. Explicit parentheses shall be required to resolve the operator-operand associations wherever an expression has a nonassociative operator to the left of an operator of the same precedence.

## 5. Constant, Variables, and Declarations

**5A. Declarations of Constants.** It shall be possible to associate identifiers with constant values of any type that is not dynamically allocated. Constants shall include both those whose values are determinable at translation time and those whose value cannot be determined until scope entry time. A translation time error shall be reported whenever a program attempts to assign to a constant valued identifier.

**5B. Declarations of Variables.** There shall be no default declarations for variables. The type of each variable must be explicitly specified in programs and shall be determinable at translation time. Variables may be of any type.

**5C. Scope of Declarations.** The intended scope of a declaration shall be determinable from the program at translation time. Scopes may be lexically embedded. Translators shall provide a warning wherever a local definition masks a more global definition. [Note that a function need not mask a more global function if they differ in name, number of parameters, or formal parameter types.]

**5D. Restrictions on Values.** Procedures, functions, types, labels, exception situations, and statements shall not be assignable to variables, computable as values of expressions, or usable as parameters to procedures or functions.

**5E. Initial Values.** There shall be no default initial values for variables. The same syntactic form shall not be used both to declare constants and to initialize variables. [Note that initialization of variables must (except for some global variables) be accomplished during execution, not translation.]

**5F. Operations on Variables.** Assignment and an implicit value access operation shall be automatically defined for each variable.

**5G. Other Declarations.** It shall be possible to associate identifiers with specifications of type and representation (including range, scale, and precision). Such identifiers may be used in declarations of variables, to specify components of elements of composite types, and in formal parameter specifications.

## 6. Control Structures

**6A. Basic Control Facility.** The built-in control mechanisms should be of minimal number and complexity and where possible shall be structured (i.e., shall have one point of entry and shall exit to a single point). Each shall provide a single capability and shall have a distinguishing syntax. Nesting of control structures shall be allowed. There shall be no control definition facility. Local scopes shall be allowed within the bodies of control statements.

**6B. Sequential Control.** There shall be a sequential control mechanism (i.e., a mechanism for sequencing statements). Explicit statement delimiters shall be required. [Note the choice between terminators and separators can be left to the user.]

**6C. Conditional Control.** There shall be conditional control structures that permit selection among alternative control paths. The selected path may depend on the value of a conditional expression, on a computed choice among labeled alternatives, or on the true condition in a set of mutually exclusive conditions. The control action must be specified for all values of the discriminating condition. [Note that only one branch will be compiled when the selected case for a conditional statement is determinable at translation time.]

**6D. Short Circuit Evaluation.** There shall be forms for short circuit conjunction and disjunction of Boolean expressions in conditional and iterative control structures.

**6E. Iterative Control.** There shall be an iterative control structure that permits a loop to have several explicit termination conditions and permits termination anywhere in the loop. Iterative control structures may be entered only at the head of the loop. [Note that when the number of iterations is zero or one and is determinable at translation time, the translator can omit any unnecessary object code.]

**6F. Loop Control Variables.** Loop control variables, if any, shall be local to the iterative control statement. Assignment shall not be allowed to control variables from the loop body. It shall be possible to iterate over sequences of integers and over elements of an enumeration type.

**6G. Explicit Control Transfer.** There shall be an explicit mechanism for control transfer (i.e., the go to). The go to shall not permit transfer of control out of declarations (including functions, procedures, and encapsulated definitions) or out of parallel control structures. It shall not permit transfer into narrower access scopes or into control structures (e.g., conditional, iterative, and parallel control structures). There shall be no control transfer mechanisms in the form of switches, designational expressions, label variables, label parameters, or alter statements.

## 7. Functions and Procedures

**7A. Function and Procedure Definitions.** Functions (which return values to expressions) and procedures (which can be called as statements) shall be definable in programs. Existing functions (including those called using infix forms) and procedures shall be extendible to new data types (i.e., overloading shall be permitted).

**7B. Recursive Definitions.** It shall be possible to define functions and procedures recursively as well as nonrecursively. If necessary the language shall restrict recursive definitions to insure that they do not add to the execution cost of other program constructs.

**7C. Scope Rules.** A reference to an identifier (other than an identifier for an exception situation) that is not declared in the most local scope shall refer to a program element that is lexically global, rather than to one that is global through the dynamic calling structure.

### 7.1. Functions

**7D. Function Declarations.** The result type for each function must be explicitly specified in the function declaration and shall be determinable at translation time. A function of two arguments may be specified as associative in its declaration. [Note that the latter requirement reduces the need for explicit parentheses.]

**7E. Restrictions on Functions.** A function may only have input parameters and may not be called in a scope that contains variables that are referenced or assigned directly or indirectly within the body of the function. [Note that this requirement guarantees that parameters to functions can be implemented safely with either value or reference passing.]

### 7.2. Parameters

**7F. Formal Parameter Classes.** There shall be three classes of formal parameters: 1) input parameters, which act as constants that are initialized to the value of corresponding actual parameters at the time of call, 2) input-output parameters, which enable access and assignment to the corresponding actual parameters, and 3) output parameters, which act as local variables whose values are transferred to the corresponding actual parameter only at the time of normal exit. In the latter two cases the corresponding actual parameter must be a variable or an assignable component of a composite type.

**7G. Parameter Specifications.** The type of each formal parameter must be explicitly specified in programs and shall be determinable at translation time. Parameters may be of any type. Range, precision, and scale specifications shall be required for each formal parameter of appropriate numeric types. A translation time error shall be reported wherever corresponding formal and actual parameters are of different types and wherever a program attempts to use a constant or an expression where a variable is required.

**7H. Formal Array Parameters.** The number of dimensions for formal array parameters must be specified in programs and shall be determinable at translation time. Determination of the subscript range for formal array parameters may be delayed until execution and may vary from call to call. Subscript ranges shall be accessible within function and procedure bodies without being passed as an explicit argument.

**7I. Restrictions to Prevent Aliasing.** Aliasing (i.e., multiple access paths to the same variable from a given scope) shall not be permitted. In particular, a variable may not be used as two output arguments in the same call to a procedure, and a nonlocal variable that is accessed or assigned within a procedure body may not be used as an output argument to that procedure.

8. Input-Output Facilities

8A. **Low Level Input-Output Operations.** There shall be a set of built-in low level input-output operations that act on physical files (e.g., input-output channels and peripheral devices). The low level operations shall be chosen to insure that all application level input-output operations can be defined within the language. They shall include operations to send control information, to receive control information, to begin transfer of data in either direction, and to wait for completion of a data transfer.

8B. **Application Level Input-Output Operations.** There shall be standard library definitions for application level input-output to logical files. These shall include operations for creating, deleting, opening, closing, reading, writing, and positioning logical files. The meaning of such operations shall depend on the general characteristics of the files or devices (e.g., on whether they are sequentially or randomly accessed), but shall not be dependent on a specific device.

8C. **Input Restrictions.** Input shall be restricted to files whose data representation is known to the translator (i.e., to files that are created and written entirely within the program or to files whose data representation is explicitly specified in the program).

8D. **Operating System Independence.** The language shall not require the presence of an operating system. The form and meaning of built-in and library definitions shall not be dependent on the operating system, if present. [Note that functions and operators of the language can be implemented as operating system calls where the operating system is compatible with the function or operator definition.]

8E. **Configuration Control.** There shall be a few low level facilities that permit programs (usually library routines) to interrogate and control the status of physical resources (e.g., memory or processors) that are managed (e.g., allocated or scheduled) by built-in features of the language. In particular it shall be possible to dynamically reassign the association between physical and logical devices, to control program overlays, and to prevent allocation and scheduling of faulty resources.

## 9. Parallel Processing

**9A. Parallel Control Structures.** There shall be a control structure for parallel processing. It shall permit a fixed number (i.e., determinable at translation time) of control paths to operate in parallel and to rejoin at a single point. There shall be an operation that is executable on any path of a parallel control structure and that causes immediate termination of the other paths (i.e., causes the other paths to move to the rejoin point).

**9B. Parallel Path Implementation.** The parallel processing facility shall be designed to minimize execution cost. In particular, parallel control paths shall be implementable with multiprocessors or with interleaved execution on a single processor.

**9C. Mutual Exclusion.** There shall be a mechanism for mutual exclusion among parallel processes. During specified portions of its execution, a parallel path shall be able to wait for and gain exclusive use of certain program declared objects and to release those objects. [Note that special asynchronous hardware and software interrupt facilities are not necessary in the language; interrupts can be treated as objects that are released upon occurrence of the interrupt.]

**9D. Real Time Constraints and Scheduling.** Constraints on the real (i.e., elapsed) time for execution of portions of control paths shall be specifiable in programs. Translators shall give warning if there is risk that time constraint will not be met. It shall also be possible to specify which paths are to be given preference in execution (in case the number of actual paths exceeds the number of available processors or the processors execute at different speeds). [Note: Such specifications provide a means to document the real time constraints of the applications, but do not specify a specific execution order among parallel paths and do not provide a safe means for mutual exclusion.]

**9E. Real Time Clock.** There shall be an accessible real time clock. It shall be possible to specify a delay on any control path for specified real time intervals. Such specifications shall be interpreted as the minimum time before continuing execution on that control path.

**9F. Simulated Time Clock.** There shall be an accessible simulated time clock. It shall be possible to delay any control path for a specified simulated time interval.

## 10. Exception Handling

10A. **Exception Handling Facility.** There shall be an exception handling mechanism for responding to unplanned error situations detected during program execution. The exception situations shall include errors detected by hardware, software errors detected during execution, error situations in built-in operations, and attempts to execute portions of programs that are not present in main memory. Exceptions should add to the execution time of programs only if they are invoked.

10B. **Software Error Situations.** The software errors detectable during execution shall include exceeding the specified range of an array subscript, exceeding the specified range of a variable, exceeding the implemented range of a variable, attempting to access an uninitialized variable, and failing to satisfy a program specified assertion. [Note that many range checks can be done during translation thereby reducing execution costs.]

10C. **Invoking Exceptions.** During any function or procedure execution it shall be possible to invoke an exception situation in the calling statement. This exception shall cause termination of the routine and an immediate transfer of control in the caller. Such exceptions must be specified in the definition of the function or procedure. Exceptions that can be invoked by built-in operations shall be given in the language definition.

10D. **Processing Exceptions.** There shall be a control structure for discriminating among the exceptions that can occur in a specified portion of a program. Exceptions that are not processed at a given function or procedure level shall cause termination of the function or procedure and shall invoke an exception in its caller. Exceptions that cause exit from parallel control structures shall terminate all paths of the parallel control structure.

10E. **Order of Exceptions.** The order in which exceptions in different parts of an expression are detected shall not be guaranteed by the language or by the translator.

10F. **Assertions.** It shall be possible to include assertions in programs. If an assertion is false when encountered during execution, it shall invoke an exception. [Note that assertions can also be used to aid optimization and maintenance.]

10G. **Suppressing Exceptions.** It shall be possible to suppress individually the detection of exceptions for software error situations. Should such a situation occur when its detection is suppressed, the consequences will be unpredictable.

## 11. Specifications of Object Representation

**11A. Data Representation.** The language shall permit but not require programs to specify the physical representation of data. These specifications shall be distinct from the logical descriptions. Specifications for the order of fields, the width of fields, the presence of "don't care" fields, the positions of word boundaries, and the object representation of atomic data shall be allowed. If object representations are not specified, they shall be determined by the translator.

**11B. Multiple Representations.** It shall be possible in programs to define more than one physical representation (e.g., packed and unpacked) for elements of a given type, and to associate a specific representation with each variable of that type. [Note that changes of representation can be accomplished through assignment.]

**11C. Machine Configuration Constants.** The language shall require the declaration of certain global constants of the object machine configuration. These shall include constants that specify the machine model, the memory size, special hardware options, the operating system if present, and peripheral equipment. Such constants shall be used to determine the object code to be generated by the translator and may also be used by the program like other constants. [Note that the user can define constants and use them as switches to control user defined compilation options.]

**11D. Configuration Dependent Specifications.** It shall be possible to use machine dependent facilities in programs. Portions of programs that depend on the characteristics of the object machine (e.g., on the machine model, special hardware options, device configuration, or operating system) shall be permitted only within branches of conditional control structures that discriminate on the object machine configuration.

**11E. Code Insertions.** For some object machines it shall be possible to write programs that include encapsulated code written in machine language or in other established programming languages. Such facilities shall be modest and shall attempt to maximize safety. The language should be designed to minimize the need for code insertions.

**11F. Optimization Specifications.** It shall be possible in programs to specify the optimization criteria to be used. It shall be possible to specify whether minimum translation costs or minimum execution costs are more important. In the latter case the user may also specify whether execution time or memory space is to be given preference. The meanings of program constructs (other than execution time and space) shall not depend on the optimizations that are applied.

## 12. Library, Separate Compilation, and Generic Definitions

**12A. Library Entries.** The language shall support the use of an external library of definitions and separately compiled segments. Library entries shall include type definitions, input-output packages, common pools of shared declarations, and application oriented software packages. The library shall be structured to allow entries to be associated with a particular application, project or user.

**12B. Separately Compiled Segments.** The language shall support the assembly of separately compiled program segments into an operational program. It shall allow definitions made in one separately compiled segment to be used in another, and shall require that such definitions and declarations be explicitly exported from the defining segment and be explicitly, but not necessarily individually, imported to the using segment. Type constraints and other program and language imposed restrictions shall be enforced across such interfaces.

**12C. Restrictions on Separate Compilation.** Separate compilation shall not change the meaning of a program. Translators shall be responsible for the integrity of object code in affected segments when any segment is modified, and shall insure that shared definitions have compatible representations in all segments. [Note: This suggests that a segment cannot be compiled until all segments from which it imports definitions and declarations, are defined.]

**12D. Generic Definitions.** It shall be possible to define functions, procedures, and types with parameters that are instantiated during translation at each call. Such parameters may be any defined identifier (including those for variables, functions, or types), an expression, or a statement. These parameters, like all other parameters, shall be evaluated in the context of the call. [Note that generic definitions generally cannot be separately compiled, but where generic definitions are implemented as closed routines, several instantiations can often share the same object code.]

## 13. Support for the Language

**13A. Defining Documents.** The language shall have a complete and unambiguous definition. It should be possible to predict the complete action of any syntactically correct program from the language definition. The language documentation shall include the syntax, semantics, and appropriate examples of each feature including those for standard library definitions. The defining documentation might point out the relative efficiency of alternative constructs.

**13B. Standards.** There will be a standard definition of the language. Procedures will be established for standards control and for certification that implementations meet the standard.

**13C. Subset and Superset Implementations.** Translators shall implement the standard definition. There shall be no subset or superset implementations. Every feature that is available to the user shall be defined in the standard, in an accessible library, or in the source program.

**13D. Translator Diagnostics.** Translators shall be responsible for reporting errors that are detectable at translation time and for optimizing object code. Translators shall do full syntax and type checking, shall check that all language imposed restrictions are met, and shall provide warnings of unusually expensive constructs. A representative set of translation time diagnostic and warning messages shall be included in the language definition.

**13E. Translator Characteristics.** Translators for the language shall be written in the language and shall be able to produce code for a variety of object machines. Where practical, the machine independent parts of translators should be separate from the code generators. Self hosting of translators is desirable, but is not required (i.e., the translator need not be able to run on all the object machines). The internal characteristics of the translator (i.e., the translation method) shall not be dictated by the language definition or standards.

**13F. Translation and Execution Restrictions.** Translators should fail to compile correct programs only when the program exceeds the resources or capabilities of the intended object machine or when the program requires more resources during the translation than are available on the host machine. Translators shall report an error when a program requires memory, devices, or special hardware that are unavailable in the object machine. Neither the language nor its translators shall impose arbitrary restrictions on language features. That is, they shall not impose restrictions on the number of array dimensions, on the size of data structures, on the size of set types, on the number of identifiers, on the length of identifiers, or on the number of nested parentheses levels unless such restrictions are dictated by the limitations of the host or object machine and are documented in user accessible manuals.

**13G. Software Tools and Application Packages.** The language shall be designed to work in conjunction with a variety of useful software tools and application support packages. These will be developed as early as possible and will include editors, interpreters, diagnostic aids, program analyzers, documentation aids, testing aids, software maintenance tools, optimizers, and application libraries. There will be a consistent user interface for these tools. Where practical software tools and aids will be written in the language. Support for the design, implementation, distribution, and maintenance of translators, software tools and aids, and application libraries will be provided independently of the individual projects that use them.

For a more detailed discussion of the DoD common language effort, the requirements background, and the relation of programming languages to the DoD software problem see:

1. Department of Defense Requirements for High Order Computer Programming Languages, "TINMAN", June 1976, or

2. IDA Paper P-1191, "A Common Programming Language for the Department of Defense --Background and Technical Requirements", David A Fisher, June 1976.

Both of the these documents have been widely distributed and both contain the December 1975 set of requirements.

# DEPARTMENT OF DEFENSE

# REQUIREMENTS FOR HIGH ORDER

# COMPUTER PROGRAMMING LANGUAGES

"TINMAN"

JUNE 1976

(2)

This "Tinman" document represents the requirements of the Department of Defense for High Order Computer Programming Languages. This is the position of the representatives of the Military Departments, as presently formulated; however, it should be regarded as a living document to be modified as requirements become more specific, as technical capabilities are better appreciated, or as expansion of the explainations becomes indicated. Comments are, therefore, actively solicited on all levels and may be addressed to the Chairman of the High Order Language Working Group:

Lt. Col. William A. Whitaker
Defense Advanced Research Projects Agency
1400 Wilson Boulvard
Arlington, Virginia 22209

And the Lord said, Behold, the people is one, and they have all one language; and this they begin to do: and now nothing will be restrained from them, which they have imagined to do.

Genesis XI 6

# I. Introduction

Like most large computer users, the United States Department of Defense has been plagued with a proliferation of high order languages and incompatible systems serving the "same" language. The DoD's problems are, in principal, no different from the rest of the computer user community; they are simply larger as is the use of computers. Further, DoD systems are often individually very large and very long lived. Because of the intimate integration of the computer resources with the rest of a large defense system, it has not been possible to change computer subsystems with frequency which might characterize a commercial operation. As a result, maintenance of the computer system is both long term and dynamic, and maintenance in many cases involves modification of the system to respond to new threats. Defense systems are often composed of interacting but independently developed subsystems, sometimes brought into existence over a period of years, all of which must be served by a common but evolving hardware base. In such an environment, the Department of Defense finds itself spending an increasingly larger fraction of its systems resources on software. High order language commonality and the resulting flexibility would provide a powerful tool for reducing the high cost of software in the DoD.

With each of the Military Departments studying this problem and making proposals for common languages, it was clear that the greatest benefit could be reaped by providing languages common across the Department of Defense. In January 1975, a DoD High Order Lanugage Working Group (HOLWG) was chartered by DDR&E with representatives from the Military Departments to investigate the requirements and specifications for programming language commonality, to compare these with existing approaches, and to recommend adoption or implementation of the necessary common language or languages. Until the matter is resolved, the DoD will not support any further implementation of new high order languages in R&D programs.

The first task of the HOL Working Group was to formulate a set of requirements consistent with the levies of the Military Departments. This effort proceeded as follows:

While there is no intent to replace the already standard COBOL and FORTRAN in their field of application, initially there was no reason to unnecessarily restrict the requirements to only those areas such as real-time applications or weapons systems where the major problems are, although this area must receive emphasis. Therefore, requirements were solicited from as broad a base as possible, to be prioritized later as required. Further, inquiries were not restricted to those programs presently using high order languages, rather, a major thrust of the effort is to provide HOL's to meet the requirements of those who are now constrained to use an assembly language for lack of a suitable HOL.

A major problem was to formulate the inquiry in such a way that meaningful requirements would result. On the one hand, one wishes to avoid broad generalities such as the obvious goals of efficiency, ease of coding, readability, etc., which, while are very real goals, are insufficiently quantified to provide any detailed guidance. On the other hand, to over specify the requirements at the level of individual language features would merely formalize past idiosyncrasies and magnify distinctions which do not represent fundamental differences. It has proved to be impossible to define rigorously the exact level requirement desired; and, therefore, a "strawman" of preliminary requirements was established to define this level by illustration. The "strawman" was not intended to be complete or consistent, rather it was deliberately provocative in order to elicit the widest possible comment. It was forwarded to the Military Departments and, by them, to their various operating organizations. In addition, it was distributed to other government agencies, to the academic community, and to industry, through industry organizations and military contractors, and by direct inquiry. A number of individuals outside the U.S. were also solicited for comments. Hundreds of individuals and organizations have had an opportunity to examine this document and provide inputs. the bulk of such inputs were positive and useful. Negative comments were almost universally based on a misunderstanding of the status of the document. Believing this to purport to be a complete and consistent set of requirements some correctly pointed out that it was no such thing.

The results of four months of such input were put together in a more concrete form, one which could then pretend to represent a fairly complete set of requirements, although still a tentative set. This document was called the"woodenman, " and it too was distributed widely. It provided a more rigorous framework for specific comment. On the basis of all inputs and the official responses from each of the Military Departments, a more complete set of requirements has evolved and is presented here as a "tinman." This document represents a set of requirements for high order computer programming language consistent with the input from the Military Services.

## II. Definition

A high order language (HOL) is one which provides compression of a computer program such that one HOL statement represents many machine language instructions.

Experienced programmers in the early 50's found that they would generate similar instruction sequences often enough that it proved useful to either set aside duplicated blocks of cards to be inserted in the program as required, or in a more automated fashion, to develop macros which might be defined once to the assembly program and thereafter invoked by shorthand notation. From such beginnings high order languages were evolved as a method for speeding up programming. The extension and development of this technique led to the more powerful concept of a higher level of programming language.

The first truly successful high order language was FORTRAN. It was developed for the IBM-704 and in the original version most of its constructs were identical to those commonly employed by machine language programmers for that machine; in some sense, it was inspired by this specific machine. But in addition, the constructs were sufficiently general that it could be adapted fairly easily to other machines and the practical reality emerged the first time of constructing a program which was machine independent, even across manufacturers. So powerful was this concept at the time that in the early 60's the use of such high order languages was heralded as automatic programming.(Annual Review of Automatic Programming, Pergamon Press, Oxford, 1960, 1961, 1963, 1964, etc.)

In 1959 a meeting was held in the Pentagon to establish a common business-oriented language, the outcome of which was COBOL (Programming Languages: History and Fundamentals, Jean E. Sammet, Prentice-Hall, Edgewood Cliffs, N.J. 1969). Other similar attempts at fairly general languages such as ALGOL, as well as those designed for a more specialized users such as CMS-2 or JOVIAL have found their somewhat smaller niche in the marketplace.

Even higher level specialized languages have come into existence such as GPSS or SIMSCRIPT for simulation programs, or ATLAS for automatic test equipment. These are problem-specific applications packages which may be generated through the use of one of the other high order language programming languages and are not in the range of the present effort.

By high order language, this report will mean a computer programming language of the general level of COBOL or FORTRAN primarily designed for programmers to communicate with computers. It is non-problem specific and should be able to exercise almost all of the capability normally used by those programming in machine language and at a reduction of an order of magnitude in the number of

statements necessary. The language is relatively machine independent, representing possible logical constructs rather than specific hardware function, and is translated by a compiler into machine code to be run on a target machine.

## III. General Goals

In this section are listed the general goals of a DoD software program. Some of the ways in which these impact on the detailed desired language characteristics set out in the next section are illustrated.

These goals, and the point of view they indicate, are generated by the chalenges of the Defense Systems Community which is concerned with what may be loosely termed Embedded Computer Resources. They have a fair overlap with, but a different emphisis from, those of the Automatic Data Processing Community.

There is no real attempt to order or weigh these goals at this point nor is there any claim that the list is complete. It is merely a very brief summary of discussions taking place during the evolution of the requirements.

COST

The overriding goal of all technology and standardization programs are to reduce the cost of Defense systems. Software is an area in which there are great opportunities in cost savings which can be brought about through actions of the DoD. Computer hardware savings are much more difficult to generate. Since the Department of Defense represents only a very small fraction of the computer hardware market, it must effectively content itself with the off-shoots of the mainstream of computer technology. On the other hand, the DoD is the main customer for software; and the major support for basic computer science and technology. It is, therefore in a position to influence the state-of-the-art and to materially change the software development process. A number of managerial initiatives to this end are presently under development. Many of them, including the setting up of libraries for software exchange, building up of a tool environment for improved software production facilities, distribution of government furnished tools, encouragement of the use of high order languages for maintenance ease, etc., depend upon, or will be facilitated by, restriction to a minimal number of common high order languages.

Much of each new software system is repetitive of processes that have been written before. A few new ideas may be added, linkages may be changed, but the bulk of the work is not new. Yet, most systems are started from scratch with little dependence on previous work other than what the individual programmers might remember. There is little high order language commonality. A new language or dialect , or at least an incompatible compiler is often generated for each project and it is impossible to sustain the creation of a truly powerful production environment.

Facilities are often locked into one particular computer because of the great expense of transferring software to the new machine. There are several proposals and on-going work in the DoD concerning the facilities for the translation of software just in a commercial-like environment. This problem and its implications may be far more serious in some of our more specialized applications. A fully supported and controlled common language facility would eliminate most of these difficulties and provide more flexibility to the managers to computer resources.

## RESPONSIVENESS AND TIMELINESS

By its very nature, software is plagued with underestimates. It represents a major fraction of the total systems capability, but is physically insignificant. There are no power requirements, no drop tests, no critical materials consumed. It is basically unsubstantive, at best, a few bits on a magnetic tape. It is by nature easy and quick to change and, by implication, it is difficult to imagine how it can be lengthy to produce. While it is easy to change, it is sometimes very hard to make it correct, as opposed to just different. As a result, estimates of the time to complete a project are often overly optimistic, sometimes so much so that the hardware is ready before the software. The cost of such delays is not just the extra man hours to bring the software into shape, but must be computed on the loss of the entire system during the period of delay. The technology then does not exist for accurately estimating software schedules other than by experience and analogy. Language can only contribute to this in that it is a part of a stable software production environment.

An actual reduction in the time to produce software would be possible if the language reduced the coding time of the project. The mere existence of the high order language normally reduces this coding time by a fair amount and certain language features may act to improve this. One must remember, however, that the coding portion of a large project normally represents only ten or fifteen percent of the total time involved and the promotion of writing ease must, therefore, be weighted with other portions of the program production cycle. A major fraction of this cycle, sometimes up to half, is involved with debugging and checking and, therefore, the thrust to more reliable code must be stronger.

Specification and design is a similarly large fraction of the effort and factors such as modularity which simplify the design are also important. The overriding consideration must, however, be the possibility of eliminating most of the effort altogether through reusing portions of both code and design which have been done before. The ability to draw upon large amounts of past work, either through libraries, previous experience, or automatic programming tools, offer the greatest possibility for payoff in timeliness and responsiveness.

It must be pointed out that this thrust towards transportability emphasizes language transportability because the elements to be transported are individual subprograms and routines, not necessarily systems concepts or coupling schemea. It is the lack of acceptance of the possibility of using previously designed and certified piece parts which separates present day hardware and software methods.

## RELIABILITY

Perhaps the most characteristically military requirement for DoD software is reliability. While all gradations of this exist in the DoD, certainly the combination of extreme complexity in systems, some almost untestable, and the life and death implications, not just of individuals but of millions, must be unique. This reliability requirement is reflected in verification, validation, and certification during the production process, as well as back-up in alternate systems, safeguards, etc. A significant portion of the software cost is tied up in this thrust, although often very subtly. Language characteristics which promote the production of reliable code must be weighted very highly.

Commonality allowing the reuse of previously certified blocks of code is also of importance. Unfortunately, the technology does not exist to automatically prove the correctness of code at a usable level. Should it eventually be possible to prove that a code will do exactly what it is intended to do and also, as in the security question, will not do anything that is not intended, the language features which make this possible would have very high priority in the DoD. Until then, perhaps the best we can do is make available facilities for investigation and experiments in this direction.

Other features promoting reliability in the DoD programming environment, would be simplicity and unambiguity of the language and readability of the constructs. It has been suggested that certain new programming techniques inherently produce better and more reliable code. A language should make it possible to use such techniques, although the brunt of enforcement must be on the programming management environment.

## MAINTAINABILITY

A major portion of Defense cost in software systems occurs during the maintenance phase. These systems are often very long lived, which is to be expected from their size and complexity and interaction with other systems. They must be kept viable for upwards of twenty years during which time major changes can take place in the threat, the employment concept, and even portions of the hardware. As a result, maintenance is often a blanket for continued development and improvement.

At the present time the statistics are somewhat deceptive in this area. On the basis of perhaps inadequate data, one would estimate that twice as much is now being spent on development as on maintenance. However, this figure is badly distorted in that the largest systems are still under development, while those in maintenance were those produced in much less ambitious times. The inventory is still growing rapidly; when it stabalizes,one might expect to see a reversal of the ratio. In most individual systems, the Operations and Maintenance costs exceed those for Development and Acquisition by several times.

Over the maintenance period, the responsibility for the program will pass through many hands both individually and corporately, between contractor and government organizations. Over time and with each transition, knowledge is lost. The more readable and definitive the program, the better knowledge will be transmitted. Transportability of the main program is usually not a question in this phase, but transportability in machine independence of the tools necessary to maintain and support that program are vital as the facilities used in this maintenance will change much more rapidly than will the main system hardware. Likewise, one of the main tools is the personnel involved. Reduction of the total number of languages with which one must work with will increase the efficiency and the reliability of the maintenance effort. Documentation of all sorts is most important in this phase. Whatever characteristics of the language that promote or support documentation, internal or external, will contribute to maintainability. A certain amount of self-documentation results just from the use of a high order language. Modularity and structured programming can further aid. Modern techniques in this area must be available.

## TRAINING

Training is a difficult factor to quantify. Formal courses are only a very small fraction of the training that a programmer gets. The bulk of it is in the form of on-job-training or simply experience. In many cases, for instance, scientific programming, the training is woefully inadequate or even non-existent. The disparity in the productivity between programmers may, in many cases, be traced to inadequate training. Better than the productivity measure would be the cost resulting from poorly written programs. Reduction of the number of languages in use would simplify the training burden.

It might be argued, particularly by expert programmers, that transitioning to a new language is fairly simple and should only take a few days of study and practice. They would then argue against paying any penalty. Unfortunately, a census of the active programmers on DoD projects, whether in-house or contract, would uncover only a small minority of such versatile experts. The average programmer is often tied to a single language and, at that, not completely proficient in its techniques and tools.

The training burden would also argue for a simple language with a minimum of unambiguous constructs. Ideally, the constructs would fail soft so that ignorance of a facility would simply deny that facility and not result in unexpected side affects.

## TRANSPORTABILITY

One of the two main technical advantages of high order languages in general is transportability, sometimes termed "machine independence." The advantage of being able to write a program in sufficiently general form that it could be compiled on to more than one computer is appealing for a number of reasons. One can transfer capabilities developed in one project to another working on different equipment, and one can expect greater lifetime for software when it can be carried on from one generation of machine to the next in the same installation. Both of these advantages are clear cut and everyone with a few years of software experience has its own examples to cite in this regard.

The language requirements associated with transportability depend on the situation. In the very early days of SHARE, the only really large-scale computer was the 704 and a great deal of interchange took place for programs written with fairly standard interfaces in machine language. This was entirely adequate for that time. The present diversity of machines requires a different technique. The use of high order language and compiling down to machine code makes a certain amount of machine independence possible but not necessarily inevitable. Any good high order language should lend itself to efficient compilers for most any machine. With appropriate linking mechanisms, programs from various languages could be put together as required. Unfortunately, the practical world intervenes and the production of good compilers for every language and every machine is not a reasonable expectation, nor is sufficient familiarity on the part of majority of programmers with a large number of languages. To make the overall software environment relatively machine independent therefore requires a minimal number of high order languages with reasonably general constructs and active support for wide availability of tools and compilers. This would further suggest that high order language compilers written in that high order language would be most appropriate.

One of the snares that has appeared in the past inhibiting transportability has been the occurrence of incompatible implementations of a single high order language. A language might be insufficiently or ambiguously specified such that different implementers making decisions in an uncontrolled fashion produce incompatible implementations. Further, each one may have his own ideas for additions to the defined language or perhaps may invoke subsets leaving out more difficult to implement or momentarily unnecessary features. We have examples of literally dozens of different implementations of a single language. These are the cruelest mockery of language commonality. Differences are sometimes so subtle as to be difficult to find and lead to undiscovered catastrophes. Such experiences give programmers an almost paranoid skepticism of claims of transportability. To make transportability a reality will, therefore, require not only a rigid and unambiguous language specification, but control or denial of subsets and supersets and rigorous certification of all implementations. It will further require the widespread availablity of such certified compilers for many machines.

The payoff from transportability could be enormous. Many programming chores are simply redoing things which have been done before. Each project usually only adds a small fraction to the sum total of knowledge. If it could draw upon past work, cost, timeliness and, reliability, would all be markedly improved. So far from reality is this today that the techniques involving libraries, etc., have not been worked out. A common high order language is simply one step, albeit an important one, towards this goal.

## READABILITY/WRITABILITY

The other major advantage initially claimed for high order languages was that they were easier to read and write, being closer to natural language than is machine code. Claims in this direction were sometimes carried to an extreme and some were promoted so highly that their advocates would have you believe that the coder need not know anything about the machine, simply write out the requirements in English. History does not always support these claims. Many such languages advocated on this basis now are the ones who have the most specialized programmers as their users. The DoD environment is sufficiently large and specialized that the allocation of personnel specifically for software is the rule anyway and there is no strong requirement to eliminate other specialists in this area. This is in contradistinction to a very small commercial firm which may have insufficient demands to justify a full-time programming staff.

Our requirement for readability and writability is therefore primarily among specialists. The desire to communicate with a computer system in natural language is an entirely different one in the DoD and involves query languages or applications programs outside the scope of this high order language effort.

Readability is clearly more important to the DoD than writability. The program is written once, but may have to be read dozens of times over over a period of years for verification, modification, etc. This is certainly true all weapons systems applications and even most of our scientific and simulation programs are very long lived. This requirement is very much different from one which might be generated, for instance, in a university environment when a program may have a life of only a few months and a single person working on it.

Ease of writing is not an inconsiderable goal but one which may be promoted through intelligent terminals, preprocessors, interactive systems, etc. The language evaluation should favor readability where conflict arises.

## EFFICIENCY

Efficiency is probably the one overriding consideration on which high order languages stand or fall. Whether it be in runtime speed or memory requirements, these quantifiable costs of using a high order language in preference to machine language or assembly code is almost universally cited as reason for rejecting HOL in a project. Often a detailed study of the matter is not accomplished, so strong do feelings run on this subject. Many changes have taken in place in the technology in recent years so as to reduce the importance of this matter, but it is still such an emotion-charged issue that it must be recognized as the principle customer concern.

Much of the pressure that existed some years ago towards efficiency is no longer relevant. In former times the cost of memory was such a large fraction of the total system expenditure that it might seem reasonable to sacrifice many other advantages in order to obtain memory efficiency. This can no longer be said to be the case. Software, particularly the maintenance portion, is becoming much more important than the hardware. Hardware costs are falling at such a rate that the change in technology over just a few months can decrease the effective cost of memory enough to offset any loss in efficiency, while the cost of going to machine code might result in a much longer delay in developing the program. Likewise, in speed there is often a penalty to pay which is invariably judged so high, often on the basis of experience with bad language implementations, that is common folklore that "HOL cannot be used in real-time applications" or that one must at least have the capability of dropping down into machine code for "time-critical" portions of the system. Yet the cost of increasing the central processor hardware capability by some appropriate amount may be far less than the software burden incurred.

While the cost arguments for HOL are valid, they do not take into account some of the realities of system procurements. It has been common in the past that the hardware configuration is determined very early in the process and the software must be made to fit. Under these conditions, particularly when life cycle cost is not an effective control, there is little choice for the project manager but to be driven to efficiency. A similar situation may arise in the later phases of system life when a new threat or other additional burden must be accommodated within existing hardware.

What measures of efficiency are reasonable? Normally, one thinks in terms of a few tens of percent. An implementation that is within ten percent of assembly language code in speed of execution or memory occupied would be difficult to distinguish, the differences in the quality of code produced by even very good programmers is probably greater than this. (In comparisons one only cites very good programmers. Many current implementations may be as good as an average programmer. Further, one must measure against code which is written in a production environment to be maintainable, etc., not using extraordinarily obscure

techniques. Unique machine dependencies may still require machine code inserts.) A thirty percent penalty might be appropriate in some situations and justified in a sufficiently detailed examination of all costs. A fifty percent degradation for going to high order language may be unacceptable for a project in which the efficiency of the project code was of significance. These figures may be overly severe from a strictly statistical view considering that the logic of a code may admit to programmer variations much greater than these, and large improvements in speed efficiency speed must be sought there. Nevertheless, there is no justification for adding on an additional excessive penalty, particularly when the high order language can be very efficient. To illustrate that an HOL can be efficient, take as an extreme example the technique of defining the program logic in a high order language which is then compiled by hand, a technique which has proved useful in the early phases of some projects. Presumably, the compilation in such a case should be very good. There are even now examples of benchmarks for which HOL compilations are better than assembly code. These may be spurious cases, but in the end this is to be expected theoretically since the HOL route gives a more practical vehicle for large-scale optimization.

One of the main difficulties with many existing high order languages is that their implementations have often been poor, particularly languages generated for single projects and languages implemented in a short time with limited resources. It is not surprising that the implementation is not as powerful as that which could be obtained if 10 or 100 times more effort could be devoted to its development and to the tools and techniques for its use. Such powerful implementations are very good and optimize better than the average programmer. Assuming a very minimal number of well-supported languages for the Department of Defense, one could devote sufficient effort to provide the tools and compilers to produce very efficient object code. It is not clear that any other language requirement mitigates against this expectation.

This particular requirement, or solution, would reinforce the thrust to commonality by making it attractive to reduce not only the number of languages supported but a number of different compiler approaches and would call for the type of control of the language which would promote large efforts for tools and compilers. A contributing technological trend is for DoD programs, particularly those which are the most critical real-time programs, is to have available very large computers on which to build the system, even though it may eventually run on a small computer. The large systems can support cross compilers which are much more powerful than those self-hosted on a smaller target machine. While this is not yet a universal procedure, it certainly provides a further opportunity to bring to bear tools to promote efficient programs. In such a case, the requirement on the high order language is to give as much information to the compiler as possible without overly restricting its options. This argument is contrary to the current practice of providing small subsets of a high order language for mini-computers. Such a subset would

only restrict the amount of information available to a compiler and would, of necessity, produce a less efficient code. The computer might produce even poorer code if forced to host on a small-target machine.

ACCEPTABILITY

No language is going to be of value unless it is used. There are many tangible and intangible influences which will determine its final acceptability. The large scale acceptability of compilers is certainly the most influential tangible asset. Perhaps the only factor that this influences in language design should be the ability to write compilers in the high order language.

That the language resembles something the customer is already familiar with is certainly going to be of considerable importance. Arbitrary decisions on formats must be take in the direction of existing DoD practice. The possibility of making translators from existing programming languages to common languages will probably never appear specifically as a requirement, since there is no intent at this time to perform such translations. However, it would be difficult to deny the attractiveness of such a possibility if it existed.

## IV. Needed Characteristics

The set of characteristics prescribed below represent a synthesis of the requirements submitted by the Military Departments and are intended to be consistent with the general goals of Section III, to be self-consistent, and to be achievable with existing computer software and hardware technology. The needed characteristics are the requirements to be satisfied by an existing, modified or new language which is selected as a Common HOL. They prescribe capabilities and properties which a common DoD language should possess but are not intended to impose any particular language features or mechanization of those capabilities. The header of each item gives a general description of the needed language characteristic while the subsequent paragraph(s) of its body provide clarification, discuss some of the implications and problems, provide the rationale behind its inclusion, and/or further detail the requirement. The entire text and not just the headers constitute the requirements.

The large number of characteristics reflects an attempt at thoroughness in dealing with the relevant issues. Similarly, the length of the discussion for many items reflects the need to resolve the ambiguities, examine the implications, and demonstrate the feasibility of the compendious statement introducing that characteristic. Because the characteristics address issues in the design, implementation, and use of the language and properties of the resulting product, there should be no correlation between the number of characteristics discussed here and the number of features in a language which satisifes these characteristics. Many of the characteristics will influence the choice of many features, and every feature will be influenced by many of the needed characteristics. Good language design is a unification process. Any language which satisfies these characteristics must be smaller and simpler than the set of issues underlying its choice.

The material reported in the last three sections (K,L,M) was generated by the Services at the same time as the technical characteristics, but is concerned with translators, support software, documentation, training, standards, application libraries, management policy, and procurement practices for the common language and its use. These issues are important. While mistakes and oversights in the technical characteristics can guarantee failure of the common language effort, success is not guaranteed no matter how technically meritorious the resulting language. Success can only be guaranteed by close attention to a variety of administrative issues, including those considered below.

Several of these issues, including those of implementation, documentation, and support will either directly or indirectly affect the acceptability of candidate languages. As with the needed technical characteristics for the common language, the issues raised here are often not resolved at the most detailed level. Until more detailed characteristics of the language come into focus there is no rationale with which to resolve all these issues in detail.

A. DATA AND TYPES
1. Typed Language
2. Data Types
3. Precision
4. Fixed Point Numbers
5. Character Data
6. Arrays
7. Records

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable, and component of composite data structures be explicitly specified in the source programs.

By the type of a data object is meant the set of objects themselves, the essential properties of those objects and the set of operations which give access to and take advantage of those properties. The author of any correct program in any programming language must, of course, know the types of all data and variables used in his programs. If the program is to be maintainable, modifiable and comprehensible by someone other than its author, the the types of variables, operations, and expressions should be easily determined from the source program. Type specifications in programs provide the redundancy necessary to verify automatically that the programmer has adhered to his own type conventions. Static type definitions also provide information at compile time necessary for production of efficient object code. Compile time determination of types does not preclude the inclusion of language structures for dynamic discrimination among alternative record formats (see A7) or among components of a union type (see E6). Where the subtype or record structure cannot be determined until run time, it should still be fully discriminated in the program text so that all the type checks can be completed at compile time.

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

These are the common data types and type generators of most programming languages and object machines. They are sufficient, when used with a data

definition facility (see E6, D6, and J1), to efficiently mechanize other desired types such as complex or vector.

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

This is a specification of what the program needs, not what the hardware provides. Machine independence, in the use of approximate value numbers (usually with floating point representation), can be achieved only if the user can place constraints on the translator and object machine without forcing a specific mechanization of the arithmetic. Precision specifications, as the minimum supported by the object code, provide all the power and guarantees needed by the programmer without unnecessarily constraining the object machine realization. Precision specifications will not change the type of reals nor the set of applicable operations. Precison specifications apply to arithmetic operations as well as to the data and therefore should be specified once for a designated scope. This permits different precisions to be used in different parts of a program. Specification of the precision will also contribute to the legibility and implementability of programs.

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.

Scaled integers are useful approximations to real numbers when dealing with exact quantity fractional values, when the object machine does not have floating point hardware, and when greater precision is required than is available with the floating point hardware. Integers will also be treated as exact quantities with a step size equal to one.

A5. Character sets will be treated as any other enumeration type.

Like any other data type defined by enumeration (see E6), it should be possible to specify the program literal and order of characters. These properties of the character set would be unalterable at run time. The definition of a character set should reflect on the manner it is used within a program and not necessarily on the

print representation a particular physical device associates with a bit pattern at run time. In general, unless all devices use the same character code, run-time translation between character sets will be required. Widely used character sets, such as, USASCII and EBCDIC will be available in a standard library. Note that access to a linear array filled with the characters of an alphabet, A, and indexed by an alphabet, B, will convert strings of characters from B to A.

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

This is general enough to permit both arrays which can be allocated at compile or load time and arrays which can be allocated at scope entry, but does not permit dynamic change to the size of constructed arrays. It is sufficient to permit allocation of space pools which the user can manage for allocation of more complex data structures including dynamic arrays. The range of subscript values for any given dimension will be a contiguous subsequence of values from an enumeration type (including integers). The preferable lower bound on the subscript range will be the initial element of an enumeration type or zero, because it often contributes to program efficiency and clarity.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

This provides all that is safe to use in CMS-2 and JOVIAL OVERLAY and in FORTRAN EQUIVALENCE. It permits hierarchically structured data of heterogeneous type, permits records to have alternative structures as long as each structure is fixed at compile time and the choice is fully discriminated at run time, but it does not permit arbitrary references to memory nor the dropping of type checking when handling overlayed structures. The discrimination condition will not be restricted to a field of the record but should be any expression.

B. OPERATIONS
1. Assignment and Reference
2. Equivalence
3. Relationals
4. Arithmetic Operations
5. Truncation and Rounding
6. Boolean Operations
7. Scalar Operations
8. Type Conversion
9. Changes in Numeric Representation
10. I/O Operations
11. Power Set Operations

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

The user will be able to declare variables for all data types. Variables are useful only when there are corresponding access and assignment operations. The user will be permitted to define assignment and access operations as part of encapsulated type definitions (see E5). Otherwise, they will be automatically defined for types which do not manage the storage for their data. (See D6 for further discussion).

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

Equivalence is an essential universal operation which should not be subject to restriction on its use. There are many useful equivalence operations for some types and a language definition cannot foresee all these for user defined types. Equivalence meaning logical identity and not bit-by-bit comparison on the internal data representation, however, is required for all data types. Proper semantic interpretation of identity requires that equality and identity be the same for atomic data (i.e., numbers, characters, Boolean values, and types defined by enumeration) and that elements of a disjoint types never be identical. Consequently, its usefulness at run time is restricted to data of the same type or to types with nonempty intersections. For floating point numbers identity will be defined as the same within the specified (minimum) precision.

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration.

Numbers and types defined by enumeration have an obvious ordering which should be available through relational operations. All six relational operations will be included. It will be possible to inhibit ordering definitions when unordered sets are intended.

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

These are the most widely used numeric operations and are available as hardware operations in most machines. Floating point operations will be precise to at least the specified precision.

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

These requirements seem obvious, particularly for floating point numbers and yet many of our existing languages truncate the most significant mantissa digits in some mixed and floating point operations.

B6. The built-in Boolean operations will include "and," "or," "not," and "nor." The operations "and" and "or" on scalars will be evaluated in short circuit mode.

Short circuit mode as used here is a semantic rather than an implementation distinction and means that "and" and "or" are in fact control operations which do not evaluate side effects of their second argument if the value of the first argument is "false" or "true," respectively. Short circuit evaluation has no disadvantages over the corresponding computational operations, sometimes produces faster executing code in languages where the user can rely on the short circuit execution, and improves the clarity and maintainability of programs by permitting expressions such

as, "i<=7 & A[i] >x" which could be erroneous were short circuit execution not intended. Note that the equivalence and nonequivalence operations (see B2) are the same as logical equivalence and exclusive-or respectively.

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

Conformability will require exactly the same number of components (although a scalar can be considered compatible with any array) and one for one compatibility in type. Correspondence will be by position in similarly shaped arrays. In many situations component by component operations are done on array elements. In fact, a primary reason for having arrays is to permit large numbers of similarly treated objects to have a uniform notation. Operations on data aggregates available directly in the source language hide the details of the sequencing and thereby, simplify the program and make more optimizations available. In addition, they permit simultaneous execution on machines with parallel processing hardware. Although component by component operations will be available for built-in composite data structures which are used to define application-oriented structures, that capability will not be automatically inherited by defined data structures. A matrix might be defined using an array, but it will not inherit the array operations automatically. Multiplication for matrices would, for example, be unnatural, confusing and inconvenient if the product operator for matrices were interpreted as a component by component operation instead of cross product of corresponding row and column vectors. Component by component operations also allow operations on character strings represented as vectors of characters and allow efficient Boolean vector operations.

Transfers between arrays or records of identical logical structure are necessary to permit efficient run time conversion from one object representation to another, as might be done when data is packed to reduce peripheral storage requirements and I/O transfer times but need to be unpacked locally to minimize processing costs.

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversions operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

Implicit type conversions which represent changes in the value of data items without an explicit indicator in the program, are not only error prone but can result in unexpected run time overhead.

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

Because ranges do not form closed systems, range validation is not possible at compile time (e.g., "I:=I+1" may be a range error). At best, the compiler might point out likely range errors. (This requirement is optional for hardware installations which do not have overflow detection).

B10. The base language will provide operations allowing programs to interact with files, channels or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

Whether the referenced "files" are real or virtual and whether they are hardware devices, I/O channels or logical files depends on the object machine configuration and on the details of its operating system if present. But in any programming system I/O operations ultimately reduce to sending or receiving data and/or control information to a file or to a device controller. These can be made accessible in a HOL in an abstract form through a small set of generic I/O operations (like "read" and "write," with appropriate device and exception parameters). Note that devices and files are similar in many respects to types, so additional language features may not be required to satisfy this requirement. This requirement, in conjunction with requirement E1, permits user definition of unique equipment and its associated I/O operations as data types within the syntactic and semantic framework provided by the generic operations.

B11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

As with any data type, power sets will be useful only if there are operations which can create, select and interrogate them. Note that this provides only a very

special class of sets but one which is very useful for computations on sets of indicators, flags, and similar devices in monitoring and control applications. More general sets if desired, must be defined using the type definition facilities.

C. EXPRESSIONS AND PARAMETERS

1. Side Effects
2. Operand Structure
3. Expressions Permitted
4. Constant Expressions
5. Consistent Parameter Rules
6. Type Agreement in Parameters
7. Formal Parameter Kinds
8. Formal Parameter Specifications
9. Variable Numbers of Parameters

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

This is a semantic restriction on the evaluation order of arguments to expressions. It provides an explicit rule (i.e., left-to-right) for order of argument evaluation, but allows the implementations to alter the actual order in any way which does not change the effect. This provides the user with a simple rule for determining the effects of interactions among argument evaluations without imposing a strict rule on compilers which are sophisticated enough to detect potential side-effects and optimize through reordering of arguments when the evaluation order does not affect the result. Control operations (e.g., conditional and iterative control structures), of course, must be exceptions to this general rule since control operations are in fact those operations which specify the sequencing and evaluation rules for their arguments.

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

Care must be taken to ensure that the operator/operand structure of expressions is not psychologically ambiguous (i.e., to guarantee that the parse implemented by the language is the same as intended by the programmer and understood by those reading the program). This kind of problem can be minimized by having few precedence levels and parsing rules by allowing explicit parentheses to specify the intended execution order, and by requiring explicit parentheses when the execution order is of significance to the result within the same precedence level (e.g., "X divided by Y divided by Z" and "X divided by Y multiplied by Z"). The user will not be able to define new operator precedence rules nor change the precedence of existing operators.

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

This is an example of not imposing arbitrary restrictions and special case rules on the user of the source language. Special mention is made here only because so many languages do restrict the form of expressions. FORTRAN, for example, has a list of seven different syntactic forms for subscript expressions, instead of allowing all forms of arithmetic expressions.

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.

The ability to write constant expressions in programs has proven valuable in languages with this capability, particularly with regard to program readability and in avoiding programmer error in externally evaluating and transcribing constant expressions. They are most often used in declarations. There is no need, however, that constant expressions impose run time costs for their evaluation. They can be evaluated once at compile time or if this is inconvenient because of incompatibilities between the host and object machines, the compiler can generate code for their evaluation at load time. In any case, the resulting value should be the same (at least within the stated precision) regardless of the object machine (see D2). Allowing constant expressions in place of constants can improve the clarity, correctness and maintainability of programs and does not impose any run time costs.

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types for exception handing, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contributes to ease of learning,

implementing and using a language; allows the user to concentrate on the programming task instead of the language; and leads to more readable, understandable, and predictable programs.

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

Type transfers hidden in procedure calls with incompatible formal and actual parameters whether intentional or accidental have long been a source of program errors and of programs which are difficult to maintain. On the other hand, there is no reason why the subscript ranges for arrays cannot be passed as part of the arguments. Some notations permit such parameters to be implicit on the call side. Formal parameters of a union type will be considered conformable to actual parameters of any of the component types.

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

The first class of data parameter acts as a constant within the procedure body and cannot be assigned to nor changed during the procedures execution; its corresponding actual parameter may be any legal expression of the desired type and will be evaluated once at the time of call. The second class of data parameter renames the actual parameter which must be a variable, the address of the actual parameter variable will be determined by (or at) the time of call and unalterable during execution of the procedure, and assignment (or reference) to the formal parameter name will assign (or access) the variable which is the actual parameter. These are the only two widely used parameter passing mechanisms for data and the many alternatives (at least 10 have been suggested) add complexity and cost to a language without sufficiently increasing the clarity or power. A language with exception handling capability must have a way to pass control and related data through procedure call interfaces. Exception handling control parameters will be specified on the call side only when needed. Actual procedure parameters will be restricted to those of similar (explicit or implicit) specification parts.

C8. Specification of the type, range, precision, dimension, scale and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.

Optional formal parameter specification permits the writing of generic procedures which are instantiated at compile time by the characteristics of their actual parameters. It eliminates the need for compile time "type" parameters. This generic procedure capability, for example, allows the definition of stacks and queues and their associated operations on data of any given type without knowing the data type when the operations are defined.

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

There are many useful purposes for procedures with variable numbers of arguments. These include intrinsic functions such as "print," generalizations of operations which are both commutative and associative such as "max" and "min," and repetitive application of the same binary operation such as the Lisp "list" operation. The use of variable number of argument operations need not and will not cause relaxation of any compile time checks, require use of multiple entry procedures allow the number of actual parameters to vary at run time, nor require special calling mechanisms. If the parameters which can vary are limited to a program specified type treated as any other argument on the call side and as elements of an array within the procedure definition, full type checking can be done at compile time. There will be not prohibition on writing a special case of a procedure for a particular number of arguments.

D. VARIABLES, LITERALS AND CONSTANTS

1. Constant Value Identifiers
2. Numeric Literals
3. Initial Values of Variables
4. Numeric Range and Step Size
5. Variable Types
6. Pointer Variables

D1. The user will have the ability to associate constant values of any type with identifiers.

The use of identifiers to represent constant values has often made programs more readable, more easily modifiable and less prone to error when the value of a constant is changed. Associating constant values with an identifier is preferable to assigning the value to a variable because it is then clearly marked in the program as a constant, can be automatically checked for unintentional changes, and often can have a more efficient object representation.

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).

Literals are needed for all atomic data types and should be provided as part of the language definition for built-in types. Regardless of the source of the data and regardless of the object machine the value of constants should be the same. For integers it should be exact and for reals it should be the same within the specified precision. Compiler writers, however, would disagree. They object to this requirement on two grounds: that it is too costly if the host and object machines are different and that it is unnecessary if they are the same. In fact, all costs are at compile time and must be insignificant compared to the life time costs resulting from object cope containing the wrong constant values. As for being unnecessary, there have been all too many cases of different values from program and data literals on the same machine because the compile time and run time conversion packages were different and imprecise.

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

The ability to initialize variables at the time of their allocation will contribute to program clarity, but a requirement to do so would be an arbitrary and sometimes costly decision to the user. Default initial values on the other hand, contribute to neither program clarity nor correctness and can be even more costly at run time. It is usually a programming error if a variable is accessed before it is initialized. It is desirable that the translator give a warning when a path between the declaration and use of a variable omits initialization. Whether a variable will be assigned is in general an unsolvable problem, but it is sometimes determinable whether assignments occur on potential paths. In the case of arrays, it it possible at compile time only to determine that some components (but not necessarily which) have been initialized. There will be provision (at user option) for run time testing for initialization.

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

Range specifications are a special form of assertion. They aid in understanding and determining the correctness of programs. They can also be used as additional information by the compiler in deciding what storage and allocation to use (e.g., half words might be more efficient for integers in the range 0 to 1000). Range specifications also offer the opportunity for the translator to insert range tests automatically for run time or debug time validation of the program logic. With the ranges of variables specified in the program, it becomes possible to perform many subscript bounds checks at compile time. These bounds, checks, however, can be only as valid as the range specifications which cannot in general be validated at compile time. Range specifications on approximate valued variables (usually with floating point implemetation) also offer the possibility of their implementation using fixed point hardware.

D5. The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.

There should not be any arbitrary restrictions on the structure of data. This permits arrays to be components of records or arrays and permits records to be components of arrays.

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

Assignment to a pointer variable will mean that the variable's name is to act as an additional label (or reference) on the datum being assigned. Assignment to a nonpointer variable will mean that the variable's name it to label a copy of the object being assigned. For data without alterable component structure or alterable component values, there is no functional difference between reference to multiple copies and multiple references to a single copy. Consequently, pointer/nonpointer will be a property only of variables for composite types and of composite array and record components Because the pointer/nonpointer property applies to all variables of a given type, it will be specified as part of the type definition. The use of pointers will be kept safe by prohibiting pointers to data structures whose allocation scope is narrower than that of the pointer variable.

Such a restriction is easily enforced at compile time using hierarchical scope rules providing there is no way to dynamically create new instances of the data type. In the latter case, the dynamically created data can be allocated with full safety using a (user or library defined) space pool which is either local (i.e., own) or global to the type definition. If variables of a type do not have the pointer property then dynamic storage allocation would be required for assignment unless their size is constant and known at the time of variable allocation. Thus, the nonpointer property will be permitted only for types (a) whose data have a structure and size which is constant in the type definition or (b) which manage the storage for their data as part of the type definition. Because pointers are often less expensive at run time than nonpointers and are subject to fewer restrictions, the specification of the nonpointer property will be explicit in programs (this is similar to the Algol-60 issue concerning the explicit specification of "value" (i.e., nonpointer) and "name" (i.e. pointer). The need for pointers is obvious in building data structures with shared or recursive substructures; such as, directed graphs, stacks, queues, and list structures. Providing pointers as absolute address types, however, produces gaps in the type checking and scope mechanisms. Type and access restricted pointers will provide the power of general pointers without their undesirable characteristics.

E. DEFINITION FACILITIES
1. User Definitions Possible
2. Consistent Use of Types
3. No Default Declarations
4. Can Extend Existing Operators
5. Type Definitions
6. Data Defining Mechanisms
7. No Free Union or Subset Types
8. Type Initialization

E1. The user of the language will be able to define new data types and operations within programs.

The number of specialized capabilities needed for a common language is large and diverse. In many cases, there is no consensus as to the form these capabilities should take in a programming language. The operational requirements dictating specific specialized language capabilities are volatile and future needs cannot always be foreseen. No language can make available all the features useful to the broad spectrum of military applications, anticipate future applications and requirements or even provide a universally "best" capability in support of a single application area. A common language needs capability for growth. It should contain all the power necessary to satisfy all the applications and the ability to specialize that power to the particular application task. A language with defining facilities for data and operations often makes it possible to add new application-oriented structures and to use new programming techniques and mechanisms through descriptions written entirely within the language. Definitions will have the appearance and costs of features which are built into the language while actually being catalogued accessible application packages. The operation definition facility will include the ability to define new infix operators (but see H2 for restrictions). No programming language can be all things to all people, but a language with data and operation definition facilities can be adapted to meet changing requirements in a variety of areas.

The ability to define data and operations is well within the state of the art. Operation definition facilities in the form of subroutines have been available in all general purpose programming languages since at least the time of early FORTRANs. Data definition facilities have been available in a variety of programming languages for almost 10 years and reached their peak with a large number of extensible languages(Stephen A. Schuman (Ed.) Proceedings of the International Symposium on Extensible Languages, SIGPLAN Notices, Vol. 6, No. 12, December 1971. Also, C. Christensen and C.J. Shaw (Ed.), Proceedings of the Extensible Language Symposium, SIGPLAN Notices 4, August 1969.) (over 30) in 1968 and shortly thereafter. A trend toward more abstract and less machine-oriented data

specification mechanisms has appeared more recently in PASCAL(Niklaus Wirth, "An Assessment of the Programming Language PASCAL, "Proceedings of the International Conference on Reliable Software 21-23 April 1973, p. 23-30). Data type definitions, with operations and data defined together, are used in several languages including SIMULA-67(Jacob Palme, "SIMULA as a Tool for Extensible Program Products, "SIGPLAN Notices, Vol. 9, No. 4, February 1974). On the other hand, there is currently much ferment as to what is the proper function and form of data type definitions.

E2. The "use" of defined types will be indistinguishable from built-in types.

Whether a type is built-in or defined within the base will not be determinable from its syntactic and semantic properties. There will be no ad hoc special cases nor inconsistent rules to interfere with and complicate learning, using and implementing the language. If built-in features and user defined data structures and operations are treated in the same way throughout the language so that the base language, standard application libraries and application programs are treated in a uniform manner by the user and by the translator, then these distinctions will grow dim to everyone's advantage. When the language contains all the essential power, when few can tell the difference between the base language and library definitions, and when the introduction of new data types and routines does not impact the compiler and the language standards, then there is little incentive to proliferate languages. Similarly, if typed definitions are processed entirely at compile time and the language allows full program specification of the internal representation, there need be no penalty in run time efficiency for using defined types.

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

As programmers, we should not expect the translator to write our programs for us (at least in the immediate future). If we somehow know that the translator's default convention is compatible with our needs for the case at hand we should still document the choice so others can understand and maintain our programs. Neither should we be able to delay definitions (possibly forget them) until they cause trouble in the operational system. This is a special case of requirement I1.

E4. The user will be able, within the source language, to extend existing operators to new data types.

When an operation is an abstraction of an existing operation for a new type or is a generalization of an existing operation, it is inconvenient, confusing and misleading to use any but the existing operator symbol or function named. The translator will not assume that commutativity of bulit-in operations is preserved by extensions, and any assumptions about the associativity of built-in or extended operations will be ignored by the translator when explicit parentheses are provided in an expression.

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

Types define abstract data objects with special properties. The data objects are given a representation in terms of existing data structures, but they are of little value until operations are available to take advantage of their special properties. When one obtains access to a type, he needs its operations as well as its data. Numeric data is needed in many applications but is of little value to any without arithmetic operations. The definable operations will include constructors, selectors, predicates, and type conversions.

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

The above list comprises a currently known set of useful definitional mechanisms for data types which do not require run time support, as do garbage collection and dynamic storage allocation. In conjunction with pointers (see D6), they provide many of the mechanisms necessary to define recursive data structures and efficient sparse data structures.

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.

Free union adds no new power not provided by discriminated union, but does require giving up the security of types in return for programmer freedom. Range and

subset specifications on variables are useful documentation and debugging aids, but will not be construed as types. Subsets do not introduce new properties or operations not available to the superset and often do not form a closed system under the superset operations. Unlike types, membership in subsets can be determined only at run time.

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

It is often necessary to do bookkeeping or to take other special action when variables of a given type are allocated or deallocated. The language will not limit the class of definable types by withholding the ability to define those actions. Initialization might take place once when the type is allocated (i.e., in its allocation scope) and would be used to set up the procedures and initialize the variables which are local to the type definition. These operations will be definable in the encapsulation housing the rest of the type definition.

F. SCOPES AND LIBRARIES
1. Separate Allocation and Access Allowed
2. Limiting Access Scope
3. Compile Time Scope Determination
4. Libraries Available
5. Library Contents
6. Libraries and Compools Indistinguishable
7. Standard Library Definitions

F1. The language will allow the user to distinguish between scope of allocation and scope of access.

The scope of allocation or lifetime of a program structure is that region of the program for which the object representation of the structure should be present. The allocation scope defines the program scope for which own variables of the structure must be maintained and identifies the time for initialization of the structure. The access scope defines the regions of the program in which the allocated structure is accessible to the program and will never be wider than the allocation scope. In some cases the user may desire that each use of a defined program structure be independent (i.e., the allocation and accessing scopes would be identical). In other cases, the various accessing scopes might share a common allocation of the structure.

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

Limited access specified in a type definition is necessary to guarantee that changes to data representations and to management routines which purportedly do not affect the calling programs are in fact safe. By rigorously controlling the set of operations applicable to a defined type, the type definition guarantees that no external use of the type can accidentally or intentionally use hidden nonessential properties of the type. Renaming separately defined programming components is necessary to avoid naming conflicts when they are used.

Limited access on the call side provides a high degree of safety and eliminates nonessential naming conflicts without limiting the degree of accessibility which can be built into programs. The alternative notion, that all declarations which are external to a program segment should have the same scope, is inconvenient and

costly in creating large systems which are composed from many subsystems because it forces global access scopes and the attendant naming conflicts on subsystems not using the defined items.

F3. The scope of identifiers will be wholly determined at compile time.

Identifiers will be declared at the beginning of their scope and multiple use of variable names will not be allowed in the same scope. Except as otherwise explicitly specified in programs, access scopes will be lexically embedded with the most local definition applying when the same identifier appears at several levels. The language will use the above lexically embedded scope rules for declarations and other definitions of identifiers to make them easy to recognize and to avoid errors and ambiguities from multiple use of identifiers in a single scope.

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

A simple base alone is not sufficient for a common language. Even though in theory such a language provides the necessary power and the capability for specialization to particular applications, the users of the language cannot be expected to develop and support common libraries under individual projects There will be broad support for libraries common to users of well recognized application areas. Application libraries will be developed as early as possible.

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

The usefulness of a language derives primarily from the existence and accessibility of specialized application-oriented data and operations. Whether a library should contain source or object code is a question of implementation efficiency and should not be specified in the definition of the source language, but the source language description will always be available. It should be remembered, however, that interfaces cannot be validated at program assembly time without some equivalent of their source language interface specifications, that object modules are machine-dependent and, therefore, not portable, that source code is often more compact than object code, and that compilers for simple languages can sometimes

compile faster than a loader can load from relocatable object programs. Library routines written on other languages will not be prohibited provided the foreign routine has object code compatible with the calling mechanisms used in the Common HOL and providing sufficient header information (e.g., parameter types, form and number) is given with the routine in Common HOL form to permit the required compile time checks at the interface.

F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

Compools have proven very useful in organizing and controlling shared data structures and shared routines. A similar mechanism will be available to manage and control access to related library definitions.

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

The convenience, ease of use and savings in production and maintenance costs resulting from using high order languages come from being able to use specialized capabilities without building them from scratch. Thus, it is essential that high level capabilities be supplied with the language. The idea is not to provide all the many special cases in the language, but to provide a few general case . hich will cover the special cases.

There is currently little agreement on standard operating system, I/O, or file system interfaces. This does not preclude support of one or more forms for the near term. For the present the important thing is that one be chosen and made available as a standard supported library definition which the user can use with confidence.

G. CONTROL STRUCTURES
1. Kinds of Control Structures
2. The Go To
3. Conditional Control
4. Iterative Control
5. Routines
6. Parallel Processing
7. Exception Handling
8. Synchronization and Real Time

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

These mechanisms, hopefully, provide a spanning set of control structures. The most appropriate operations in several of these areas is an open question. For the present, the choice will be a spanning set of composable control primitives each of which is easily mapped onto object machines and which does not impose run time charges when it is not used. Whether parallel processing is real (i.e., by multiprocessing) or is synthesized on a single sequential processor, is determined by the object machine, but if programs are written as if there is true parallel processing (and no assumption about the relative speeds of the processors) then the same results will be obtained independent of the object environment.

It is desirable that the number of primitive control structures in the language be minimized, not by reducing the power of the language, but by selecting a small set of composable primitives which can be used to easily build other desired control mechanisms within programs. This means that the capabilities of control mechanisms must be separable so that the user need not pay either program clarity or implementation costs for undesired specialized capabilities. By these criteria, the Algol-60 "FOR" would be undesirable because it imposes the use of a loop control variable, requires that there be a single terminal condition and that the condition be tested before each iteration. Consequently, "FOR" cannot be composed to build other useful iterative control structures (e.g., FORTRAN "DO"). The ability to compose control structures does not imply an ability to define new control operations and such an ability to define new control operations, and such an ability is in conflict with the limited parameter passing mechanisms of C7.

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.

The "GO TO" is a machine level capability which is still needed to fill in any gaps which might remain in the choice of structured control primitives, to provide compatibility for translitterating programs written in older languages, and because of the wide familiarity of current practitioners with its use. The language should not, however, impose unnecessary costs for its presence. The "GO TO" will be limited to explicitly specified program labels at the same scope level. Neither should the language provide specialized facilities which encourage its use in dangerous and confusing ways. Switches, designational expressions, label variables, label parameters and numeric labels are not desired. Switches here refer to the unrestricted switches which are generalizations of the "GO TO" and not to case statements which are a general form for conditionals(see G3). This requirements should not be interpreted to conflict with the specialized form of control transfer provided by the exception handling control structure of G7.

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

The conditional control operations will be fully partitioned (e.g., an "ELSE" clause must follow each "IF THEN") so the choice is clear and explicit in each case. There will be some general form of conditional which allows an arbitrary computation to determine the selected situation (e.g., Zahn's device(Donald E. Knuth, "Structured Programming with go to Statements," ACM Computer Surveys, Vol. 6, No. 4, December 1974) provides a good solution to the general problem). Special mechanisms are also needed for the more common cases of the Boolean expression (e.g., "IF THEN ELSE") and for value or type discrimination (e.g., "CASE" on one of a set of values or subtype of a union).

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables t be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

In its most general form, a programmed loop is executed repetitively until some computed predicate becomes true. There may be more than one terminating predicate, and they might appear anywhere in the loop. Specialized control structures (e.g., "WHILE DO") have been used for the common situation in which the

termination conditions precedes each iteration. The most common case is termination after a fixed number of iterations and a specialized control structure should be provided for that purpose (e.g., FORTRAN "DO" or Algol-60 "FOR"). A problem which arises in many programming languages is that loop control variables are global to the iterative control and thus, will have a value after loop termination, but that value is usually an accident of the implementation. Specifying the meaning of control variables after loop termination in the language definition resolves the ambiguity but must be an arbitrary decision which will not aide program clarity or correctness, and may interfere with the generation of efficient object code. Loop control variables are by definition variables used to control the repetitive execution of a programmed loop and as such will be local to the loop body, but at loop termination it will be possible to pass their value (or any other computed value) out of the loop, conveniently and efficiently.

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

Recursion is desirable in many applications because it contributes directly to their elegance and clairty and simplifies proof procedures. Indirectly, it contributes to the reliability and maintainability of some programs. Recursion is required in order to avoid unnecessarily opaque, complex and confusing programs when operating on recursive data structures. Recursion has not been widely used in DoD software because many programming languages do not provide recursion, practitioners are not familiar with its use, and users fear that its run time costs are to high. Of these, only the run time costs would justify its exclusion from the language.

A major run time cost often attributed to recursion is the need for the presence of a set of "display" registers which are used to keep track of the addresses of the various levels of lexically imbedded environments and which must be managed and updated at run time. The display, however, is necessary only in programs in which routines access variables which are global to their own definition, but local to a more global recursive procedure. This possibility can easily be removed by prohibiting the definition of procedures within the body of a recursive procedure. The utility of such a combination of capabilities is very questionable, and this single restriction will eliminate all added execution costs for nonrecursive procedures in programs which contain recursive procedures.

As with any other facility of the language, routines should be implemented in the most efficient manner consistent with their use and the language should be designed so that efficient implementations are possible. In particular, the most possible regardless of whether the language or even the program contains recursive

procedures. When any routine makes a procedure call as its last operation before exit (and this is quite common for recursive routines) the implementation might use the same data area for both routines, and do a jump to the head of the called procedure thereby saving much of the overhead of a procedure call and eliminating a return. The choice between recursive and nonrecursive routines involves trade-offs. Recursive routines can aid program clarity when operating on recursive data, but can detract from clarity when operating on iterative data. They can increase execution time when procedure call overhead is greater than loop overhead and can decrease execution times when loop overhead is the more expensive. Finally, program storage for recursive routines is often only a small fraction of that for a corresponding iterative procedure, but the data storage requirements are often much larger because of the simultaneous presence of several activations of the same procedure.

G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

A parallel processing capability is essential in embedded computer applications. Programs must send data to, receive data from, and control many devices which are operating in parallel. Multiprogramming (a form of pseudo paralell processing) is necessary so that many programs within a system can meet their differing real time constraints. The parallel processing capability will minimally provide the ability to define and call parallel processing and the ability to gain exclusive use of system resources in the form of data structures, devices and pseudo devices. This latter ability satisfies one of the two needs for synchronization of parallel processes. The other is required in conjunction with real time constraints (see G8).

The parallel processing capability will be defined as true parallel (as opposed to coroutine) primitives, but with the understanding that in most implementations the object computer will have fewer processors (usually one) than the number of parallel paths specified in a program. Interleaved execution in the implementation may be required.

The parallel processing features of the language should be selected to eliminate any unnecessary overhead associated with their use. The costs of parallel processes are primarily in run time storage management. As with recursive routines most accessing and storage management problems can be eliminated by prohibiting complex interactions with other language facilities where the combination has little if any utility. In particular, it will not be possible to define a parallel routine within the

body of a recursive routine and it will not be possible to define any routine including parallel routines within the body of those parallel routines which can have multiple simultaneous activations. If the language permits several simultaneous activations of a given parallel process then it might require the user to give a upper bound on the number which can exist simultaneously. The latter requirement is reasonable for parallel processes because it is information known by the programmer and necessary to the maintainer, because parallel processes cannot normally be stacked, and because it is necessary for the compilation of efficient programs.

G7. The exception handing control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

It is essential in many aplications that there be no program halts beyond the user's control. The user must be able to specify the action to be taken on any exception situation which might occur within his program. The exception handling mechanism will be parameterized so data can be passed to the recovery point. Exception situations might include arithmetic overflow, exhaustion of available space, hardware errors, any user defined exceptions and any run time detected programming error.

The user will be able to write programs which can get out of an arbitrary nest of control and intercept it at any embedding level desired. The exception handling mechanism will permit the user to specify the action to be taken upon the occurrence of a designated exception within any given access scope of the program. The transfers of control will, at the users option, be either forward in the program (but never to a narrower scope of access or out of a procedure) or out of the current procedure through its dynamic (i.e., calling structure. The latter form requires an exception handling formal parameter class (see C7).

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

When parallel or pseudo parallel paths appear in a program it must be possible to specify their relative priorities and to synchronize their executions. Synchronization can be done either through exclusive access to data (see G6) or through delays terminated by designated situations occurring within the program.

These situations should include the elapse of program specified time intervals, occurrence of hardware interrupts and those designated in the program. There will be no implicit evaluation of program determined situations. Time delays will be program specifiable for both real and simulated times.

H. SYNTAX AND COMMENT CONVENTIONS
1. General Characteristics
2. No Syntax Extensions
3. Source Character Set
4. Identifiers and Literals
5. Lexical Units and Lines
6. Key Words
7. Comment Conventions
8. Unmatched Parentheses
9. Uniform Referent Notation
10. Consistency of Meaning

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

Clarity and readability of programs will be the primary criteria for selecting a syntax. Each of the above points can contribute to program clarity. The use of free format, mnemonic identifiers and conventional forms allows the programmer to use notations which have their familiar meanings to put down his ideas and intentions in the order and form that humans think about them, and to transfer skills he already has to the solution of the problem at hand. A simple uniform language reduces the number of cases which must be dealt with by anyone using the language. If programs are difficult for the translator to parse they will be difficult for people. Similar things should use the same notations with the special case processing reserved for the translator and object machine. The purpose of mnemonic identifiers and key words is to be informative and increase the distance between lexical units of programs. This does not prevent the use of short identifiers and short key words.

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences.

If the user can change the syntax of the language, then he can change the basic character and understanding of the language. The distinction between

semantic extensions and syntactic extensions is similar to that between being able to coin new words in English or being able to move to another natural language. Coining words requires learning those new meanings before they can be used, but at the same time increases the power the language for some application areas. Changing the grammar, (e.g., Franglais, the use of French grammar with interspersed English words) however, undermines the basic understanding of the language itself, changes the mode of expression, and removes the commonalities which obtain between various specializations of the language. Growth of a language through definition of new data and operations and the introduction of new words and symbols to identify them is desirable, but there should be no provision for changing the grammatical rules of the language. This requirement does not conflict with E4 and does not preclude associating new meanings with existing infix operators.

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

A common language should use notations and a character set convenient for communicating algorithms, programs, and programming techniques among its users. On the other hand, the language should not require special equipment (e.g., card readers and printers) for its use. The use of the 64 character ASCII subset will make the language compatible with the federal information processing standard 64 character set, FIPS-1, which has been adopted by the U.S.A. Standard code for Information Interchange (USASCII). The language definition will specify the translation from the publication language into the restricted character set.

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

Lexical units of the language should be defined in a simple uniform and easily understood manner. Some possible break characters are the space(W. Dijkstra, coding examples in Chapter I, "Notes in Structured Programming," in Structured Programming by O.-J. Dahl, E. W. Dijkstra and C.A.R. Moore, Academic Press, 1972. & Thomas A. Standish, "A Structured Program to Play Tic-Tac-Toe," notes for Information and Computer Science 3 course at Univ. of California-Irvine, October 1974) (i.e., any number of spaces or end-of-line), the underline and the tilde. The space cannot be used if identifiers and user defined infix operators are lexically indistinguishable, but in such a case the formal grammar for the language would be ambiguous (see H1). A literal break character contributes to the readability of

programs and makes the entry of long literals less error prone. With a space as a break character one can enter multipart (i.e., more than one lexical unit) identifiers such as "REAL TIME CLOCK" or long literals, such as, "3.14159 26535 89793." Use of a break can also be used to guarantee that missing quote brackets on character literals do not cause errors which propagate beyond the net end-of-line. the language should require separate quoting of each line of a long literal: "This is a long" "literal string".

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

Many elementary input errors arise at the end of lines. Programs are input on line oriented media but the concept of end-of-line is foreign to free format text. Most of the error prone aspects of end-of-line can be eliminated by not allowing lexical units to continue over lines. The sometimes undesirable effects of this restriction can be avoided by permitting identifiers and literals to be composed from more than one lexical unit (see H4) and by evaluating constant expressions at compile time (see C4).

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

By key words of the language are meant those symbols and strings which have special meaning in the syntax of programs. They introduce special syntactic forms such as are used for control structures and declarations or the are used as infix operators, or as some form of parenthesis. Key words will be reserved, that is unusable as identifiers, to avoid confusion and ambiguity. Key words will be few in number because each new key word introduces another case in the parsing rules and thereby adds to complexity in understanding the language, and because large numbers of key words inconvenience and complicate the programmer's task of choosing informative identifiers. Key words should be concise, but being information is more important than being short. A major exception is the key word introducing a comment; it is the comment and not its key word which should do the informing. Finally, there will be no place in a source language program in which a key word can be used in place of an identifier. That is, functional form operations and special data items built into the language or accessible as a standard extension will not be treated as key words but will be treated as any other identifier.

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

These are all obvious points which will encourage the use of comments in programs and avoid their error prone features in some existing languages. Comments anywhere reasonable in a program will not be taken to mean that they can appear internal to a lexical unit, such as, an identifier, key word, or between the opening and closing brackets of a character string. One comment convention which nearly meets these criteria is to have a special quote character which begins comments and with either the quote or an end-of-line ending each comment. This allows both embedded and line-oriented comments.

H8. The language will not permit unmatched parentheses of any kind.

Some programming languages permit closing parentheses to be omitted. If, for example, a program contained more "BEGINs" than "ENDs" the translator might insert enough "ENDs" at the end of the program to make up the difference. This makes programs easier to write because it sometimes saves writing several "ENDs" at the end of programs and because it eliminates all syntax errors for missing "ENDs." Failure to require proper parentheses matching makes it more difficult to write correct programs. Good programming practice requires that matching parentheses be included in programs whether or not they are required by the language. Unfortunately, if they are not required by the language then there can be no syntax check to discover where errors were made. The language will require full parentheses matching. This does not preclude syntactic features such as "case x of s1, s2...sn end case" in which "end" is paired with a key word other than "begin." Nor does it alone prohibit open forms such as "if-then-else-."

H9. There will be a uniform referent notation.

The distinction between function calls and data reference is one of representation, not of use. Thus, there will be no language imposed syntactic distinction between function calls and data selection. If, for example, a computed function is replaced by a lookup table there should be no need to change the calling program. This does not preclude the inclusion of more than one referent notation.

H10. No language defined symbols appearing in the same context will have essentially different meanings.

This contributes to the clarity and uniformity of programs, protects against psychological ambiguity and avoids some error prone features of extant languages. In particular, this would exclude the use of = to imply both assignment and equality, would exclude conventions implying that parenthesized parameters have special semantics (as with PL/1 subroutines), and would exclude the use of an assignment operator for other than assignment (e.g., left hand side function calls). It would not, however, require different operator symbols for integer, real or even matrix arithmetic since these are in fact special cases of the same abstract operations and would allow the use of generic functions applicable to several data types.

I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS
1. No Defaults in Program Logic
2. Object Representation Specifications Optional
3. Compile Time Variables
4. Conditional Compilation
5. Simple Base Language
6. Translator Restrictions
7. Object Machine Restrictions

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

The only alternative is implementation dependent defaults with the translator determining the meaning of programs. What a program does, should be determinable from the program and the defining documentation for the programming language. This does not require that binding of all program properties be local to each use. Quite the contrary, it would, for example, allow automatic definition of assignment for all variables or global specification of precision. What it does require is that each decision be explicit: in the language definition, global to some scope, or local to each use. Omission of any selection which affects the program logic will be treated as an error by the translator.

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

The language should be oriented to provide a high degree of management control and visibility to programs and toward self-documenting programs with the programmer required to make his decisions explicit. On the other hand, the programmer should not be forced to overspecify his programs and thereby cloud their logic, unnecessarily eliminate opportunities for optimization, and misrepresent arbitrary choices as essential to the program logic. Defaults will be allowed, in fact, encouraged in don't care situations. Such defaults will include data representations (see J4), open vs. closed subroutine calls (see J5), and reentrant vs. nonreentrant code generation.

I3. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

When a language has different host and object machines and when its compilers can produce code for several configurations of a given machine, the programmer should be able to specify the intended object machine configuration. The user should have control over the compile time variables used in his program. Typically they would be associated with the object computer model, the memory size, special hardware options, the operating system if present, peripheral equipment or other aspects of the object machine configuration. Compile time variables will be set outside the program, but available for interrogation within the program (see I4 and C4).

I4. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

An environmental inquiry capability permits the writing of common programs and procedures which are specialized at compile time by the translator as a function of the intended object machine configuration or of other compile time variables (see I3). This requirement is a special case of evaluation of constant expressions at compile time (see C4). It provides a general purpose capability for conditional compilation.

I5. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capablity not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

The capabilities available in any language can be partitioned into two groups, those which are definable within the base and those which provide an essential primitive capability of the language. The smaller and simpler the base the easier the language will be to learn and use. A clearly delineated base with features not in the base defined in terms of the base, will improve the ease and efficiency of learning, implementing and maintaining the language. Only the base need be implemented to make the full source language capability available.

Base features will provide relatively low level general purpose capabilities not yet specialized for particular applications. There will be no prohibition on a translator incorporating speciaized optimizations for particular extensions. Any extension provided by a translator will, however, be definable within the base language using the built-in definition facilities. Thus, programs using the extension will be translatable by any compiler for the language but not necessarily with the same object efficiency.

I6. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

Limits on the number of array dimensions, the length of identifiers, the number of nested parentheses levels in expressions, or the number of identifiers in programs are determined by the translator and not by the object machine. Ideally, the limits should be set so high that no program (save the most abrasive) encounters the limits. In each case, however: (a) some limit must be set, (b) whatever the limit, it will impose on some and therefore must be known by the users of the translator, (c) letting each translator set its own limits means that programs will not be portable, (d) setting the limits very high requires that the translator be hosted only on large machines and (e) quite low limits do not impose significantly on either the power of the language or the readability of programs. Thus, the limits should be set as part of the language definition. They should be small enough that they do not dominate the compiler and large enough that they do not interfere with the usefulness of the language. If they were set at say the 99 percent level based on statistics from existing DoD computer programs the limits might be a few hundred for numbers of identifiers and less than ten in the other cases mentioned above.

I7. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

Limits on the amount of run time storage, access to specialized peripheral equipments, use of special hardware capabilities and access to real time clocks are dependent on the object machine and configuration. The translator will report when a program exceeds the resources or capabilities of the intended object machine but will not build in arbitrary limits of its own.

## J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

1. Efficient Object Code
2. Optimizations Do Not Change Program Effect
3. Machine Language Insertions
4. Object Representation Specifications
5. Open and Closed Routine Calls

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

The base language and library definitions might contain features and capabilities which are not needed by everyone, or at least, not be everyone all the time. The language should not force programs to require greater generality than they need. When a program does not use a feature or capability it should pay no run time cost for the feature being in the language or library. When the full generality of a feature is not used, only the necessary (reduced) cost should be paid. Where possible, language features (such as, automatic and dynamic array allocation, process scheduling, file management and I/O buffering) which require run time support packages should be provided as standard library definitions and not as part of the base language . The user will not have to pay time and space for support packages he does not use. Neither will there be automatic movement of programs or data between main store and backing shore which is not under program control (unless the object machine has virtual memory with underlying management beyond the control of all its users). Language features will result in special efficient object codes when their full generality is not used. A large number of special cases should compile efficiently. For example, a program doing numerical calculations on unsubscripted real variables should produce code no worse than FORTRAN. Parameter passing for single argument routines might be implemented much less expensively than multiple argument routines.

One way to reduce costs for unneeded capabilities is to have a base language whose data structures and operations provide a single capability which is composable and has a straight-forward implementation in the object code of conventional architecture machines. If the base language components are easily composable they can be used to construct the specialized structures needed by specific applications, if they are simple and provide a single capability they will not force the use of unneeded capabilities in order to obtain needed capabilities, and if they are compatible with the features normally found in sequential uniprocessor digital computers with random access memory they will have near minimum or at least low cost implementation on many object machines.

J2. Any optimizations performed by the translator will not change the effect of the program.

More simply, the translator cannot give up program reliability and correctness, regardless of the excuse. Note that for most programming languages there are few known safe optimizations and many unsafe ones. The number of applicable safe optimizations can be increased by making more information available to the compiler and by choosing language constructs which allow safe optimizations. This requirement allows optimization by code motion providing that motion does not change the effect of the program.

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

It is difficult to be enthusiastic about machine language insertions. They defeat the purpose of machine independence constrain the implementation techniques complicate the diagnostics, impair the safety of type checking, and detract from the reliability, readability, and modifiability of programs. The use of machine language insertions is particularly dangerous in multiprogramming applications because they impair the ability to exclude, "a priori," a large class of time-dependent bugs. Rigid enforcement of scope rules by the compiler in real time applications is a powerful tool to ensure that one sequential process will not interfere with others in an uncontrolled fashion. Similarly, when several independent programs are executed in an interleaved fashion, the correct execution of each may depend on the others not having improperly used machine language insertions.

Unfortunately machine language insertions are necessary for interfacing special purpose devices, for accessing special purpose hardware capabilities, and for certain code optimizations on time critical paths. Here we have an example of Dijkstra's dilemma in which the mismatch between high level language programming and the underlying hardware is unacceptable and there is no feasible way to reject the hardware. The only remaining alternative is to "continue bit pushing in the old way, with all the known ill effects." Those ill effects can, however, be constrained to the smallest possible perimeter in practice if not in theory. The ability to enter machine language should not be used as an excuse to exclude otherwise needed facilities from the HOL; the abstract description of programs in the HOL should not require the use of machine language insertions. The semantics of machine language insertions will be determinable from the HOL definition and the object machine description alone and not dependent on the translator characteristics Machine language insertions will be encapsulated so they can be easily recognized and so that it is clear which variables and program identifiers are accessed within the insertion. The machine language insertions will be permitted only within the body of compile

time conditional statements (see I4) which depend on the object machine configuration (see I3). They will not be allowed interspersed with executable statements of the source language.

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

It is often necessary to give detailed specifications of the object data representations to obtain maximum density for large data files to meet format requirements imposed by the hardware of peripheral equipment, to allow special optimizations on time critical paths, or to ensure compatibility when transferring data between machines.

It will be possible to specify the order of the fields, the width of fields, the presence of don't care fields, and the position of word boundaries. It will be possible to associate source language identifiers (data or program) with special machine addresses. The use of machine dependent characteristics of the object representation will be restricted as with machine dependent code (see J3). When multiple fields per word are specified the compiler may have to generate some form of shift and mask operations for source program references and assignments to those variables (i.e., fields). As with machine-language insertions, object data specifications should be used sparingly and the language features for their use must be Spartan, nor grandiose.

If the object representation of a composite data object is not specified in the source program, there will be no specific default guaranteed by the translator. The translator might, for example, attempt to minimize access time and/or memory space in determining the object representation. It might, depending on the object machine characteristics, assign variables and fields of records to full words, but assign array elements to the smallest of bits, bytes, half words, words or exact multiple words permitted by the logical description.

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.

The use of inline open procedures can reduce the run time execution costs significantly in some cases. There are the obvious advantages in eliminating the parameter passing, in avoiding the saving of return marks, and in not having to pass arguments to and from the routine. A less obvious, but often more important advantage in saving run time costs is the ability to execute constant portions of routines at compile time and, thereby, eliminate time and space for those portions of the procedure body at run time. Open routine capability is especially important for machine language insertions.

The distinction between open and closed implementation of a routine is an efficiency consideration and should not affect the function of the routine. Thus, an open routine will differ from a syntax macro in that (a) its global environment is that of its definition and not that of its call and (b) multiple occurrences of a formal value (i.e., read only) parameter in the body have the same value. If a routine is not specified as either open or closed the choice will be optimal as determined by the translator.

K. PROGRAM ENVIRONMENT
1. Operating System Not Required
2. Program Assembly
3. Software Development Tools
4. Translator Options
5. Assertions and Other Optional Specifications

K1. The language will not require that the object machine have an operating system. When the object machine does have an operating system or executive program, the hardware/operating system combination will be interpreted as defining an abstract machine which acts as the object machine for the translator.

A language definition cannot dictate the architecture of existing object machines whether defined entirely in hardware or in a hardware/software combination. It can provide a source language representation of all the needed capabilities and attempt to choose these so they have an obvious and efficient translation in the object machines.

K2. The language will support the integration of separately written modules into an operational program.

Separately written modules in the form of routines and type definitions are necessary for the management of large software efforts and for effective use of libraries. The user will be able to cause anything in any accessible library to be inserted into his program. This is a requirement for separate definition but not necessarily for separate compilation. The decision as to whether separately defined program modules are to be maintained in source or object language form is a question of implementation efficiency, will be a local management option and will not be imposed by the language definition. The trade-offs involved are complicated by other requirements for type checking of parameters (see C6), for open subroutines (see J5), for efficient object representations (see J1), and for constant expression evaluation at compile time (see C4). In general, separate compilation increases the difficulty and expense of the interface validations needed for program safety and reliability and detracts from object program efficiency by removing many of the optimizations otherwise possible at the interfaces, but at the same time it reduces the cost and complexity of compilation.

K3. A family of programming tools and aids in the form of support packages including linkers, loaders and debugging systems will be made available with the language and its translators. There will be a consistent easily used user interface for these tools.

The time has passed in which a programming language can be considered in isolation from its programming environment. The availability of programming tools which need not be developed and/or supported by individual projects is a major factor in the acceptability of a language. There is no need to restrict the kinds or form of support software available in the programming environment, and continued development of new tools should be encouraged and made available in a competitive market. It is, however, desirable that tools be developed in their own source language to simplify their portability and maintainability.

K4. A variety of useful options to aid generation, test, documentation and modification of programs will be provided as support software available with the language or as translator options. As a minimum these will include program editing, post- mortem analysis and diagnostics, program reformating for standard indentations, and cross-reference generation.

There will be special facilities to aid the generation, test, documentation and modification of programs. The "best" set of capabilities and their proper form is not currently known. Since nonstandard translator options and availability of nonstandard software tools and aids do not adversely affect software commonality, the language definition and standards will not dictate arbitrary choices. Instead, the development of language associated tools and aids will be encouraged within the constraint of implementing and supporting the source language as defined. Tools and debugging aids will be source language oriented.

Some of the translator options which have been suggested and may be useful include the following. Code might be compiled for assertions which would give run time warnings when the value of the assertion predicate is false. It might provide run time tracing of specified program variables. Dimensional analysis might be done on units of measure specifications. Special optimizations might be invoked. There might be capability for timing analysis and gathering run time statistics. There might be translator supplied feedback to provide management visibility regarding progress and conformity with local conventions. The user might be able to inhibit code generation. There might be facilities for compiling program patches and for controlling access to language features. The translator might provide a listing of the number of instructions generated against corresponding source inputs and/or an estimate of their execution times. It might provide a variety of listing options.

K5. The source language will permit inclusion of assertions, assumptions, axiomatic definitions of data types, debugging specifications;, and units of measures in programs. Because many assertional methods are not yet powerful enough for practical use, nor sufficiently well developed for standardization, they will have the status of comments.

There are many opinions on the desirability, usefulness, and proper form for each of these specifications. Better program documentation is needed and specifications of these kinds may help. Specifications also introduce the possibility of automated testing, run time verification of predicates, formal program proofs, and dimensional analysis. The language will not prohibit inclusion of these forms of specification if and when they become available for practical use in programs. Assertions, assumptions, axiomatic definitions and units of measure in source language programs should be enclosed in special brackets and should be treated as interpreted comments -- comments which are delimited by special comment brackets and which may be interpreted during translation or debugging to provide units analysis, verification of assertions and assumptions, etc.--but whose interpretation would be optional to translator implementations.

L. TRANSLATORS
1. No Superset Implementations
2. No Subset Implementations
3. Low-Cost Translation
4. Many Object Machines
5. Self-Hosting Not Required
6. Translator Checking Required
7. Diagnostic Messages
8. Translator Internal Structure
9. Self-Implementable Language

L1. No implementation of the language will contain source language features which are not defined in the language standard. Any interpretation of a language feature not explicitly permitted by the language definition will be forbidden.

This guarantees that use of programs and software subsystems will not be restricted to a particular site by virtue of using their unique version of the language. It also represents a commitment to freezing the source language, inhibiting innovations and growth in the form of the source language, and confining the base language to the current state of the art in return for stability, wider applicability of software tools, reusable software, greater software visibility, and increased payoff for tool building efforts. It does not, however, disallow library definition optimizations which are translator unique.

L2. Every translator for the language will implement the entire base language. There will be no subset implementations of the base language.

If individual compilers implement only a subset of the language, the there is no chance for software commonality. If a translator does not implement the entire language, it cannot give its users access to standard supported libraries or to application programs implemented on some other translator. Requiring that the full language be implemented will be expensive only if the base language is large, complex, and nonuniform. The intended source language product from this effort is a small simple uniform base language with the specialized features, support packages, and complex features relegated to library routines not requiring direct translator support. If simple low cost translators are not feasible for the selected language, then the language is too large and complex to be standardized and the goal of language commonality will not be achievable.

L3. The translator will minimize compile time costs. A goal of any translator for the language will be low cost translation.

Where practical and beneficial the user will have control over the level of optimization applied to his programs. The programmer will have control over the tradeoffs between compile time and run time costs. The desire for small efficient translators which can be hosted by machines with limited size and capability should influence the design of the base language against inclusion of unnecessary features and towards systematic treatment of features which are included. The goal will be effective use of the available machines both in object execution and translation and not maximal speed of translation.

Translation costs depend not only on the compiler but the language design. Both the translator and the language design will emphasize low cost translation, but in an environment of large and long-lived software products this will be secondary to requirements for reliability and maintainability. Language features will be chosen to ensure that they do not impose costs for unneeded generality and that needed capabilities can be translated into efficient object representations. This means that the inherent costs of specific language features is the context of the total language must be understood by the designers, implementers and users of the language. One consequence should be that trivial programs compile and run in trivial time. On the other hand, significant optimization is not expected from a minimal cost translator.

L4. Translators will be able to produce code for a variety of object machines. The machine independent parts of translators might be built independent of the code generators.

There is currently no common widely used computer in the DoD. There are at least 250 different models of commercial machines in use in DoD with many more specialized machines. A common language must be applicable to a wide variety of models and sizes of machines. Translators might be written so they can produce object code for several machines. This reduces the proliferation of translators and makes the full power of an existing translator available at the cost of producing an additional code generator.

L5. The translator need not be able to run on all the object machines. Self-hosting is not required, but is often desirable.

The DoD operational programming environment includes many small machines which are unable to support adequately the design, documentation, test, and

debugging aids necessary for the development of timely, reliable or efficient software. Large machine users should not be penalized for the restrictions of small machines when a common language is used. On the other hand, the size of machines which can host translators should be kept as small as possible by avoiding unnecessary generality in the language.

L6. The translator will do full syntax checking, will check all operations and parameters for type compatibility and will verify that all language imposed semantic restrictions on the source programs are met. It will not automatically correct errors detected at compile time.

The purpose of source language redundancy and avoidance of error prone language features is reliability. The price is paid in programmer inconvenience in having to specify his intent in greater detail. The payoff comes when the translator checks that the source program is internally consistent and adheres to its authors' stated intentions. There is a clear trade-off between error avoidance and programming ease; surveys conducted in the Services show that the programmers as well as managers will opt for error avoidance over ease when given the choice. The same choice is dictated by the need for well documented maintainable software.

L7. The translator will produce compile time explanatory diagnostic error and warning messages. A suggested set of error and warning situations will be provided as part of the language definition.

The translator will attempt to provide the maximal useful feedback to its user. Diagnostic messages will not be coded but will be explanatory and in source language terms. Translators will continue processing and checking after errors have been found but should be careful not to generate erroneous messages because of translator confusion. The translator will always produce correct code; when source programs errors are encountered by the translator or referenced program structures omitted, the compiler will produce code to cause a run time exception condition upon any attempt to execute those parts of the program. Warnings will be generated when a source language construct is exceptionally expensive to implement on the specified object machine. A suggested set of diagnostic messages provided as part of the language definition contributes to commonality in the implementation and use of the language. The discipline of designing diagnostic messages keyed to the design may also uncover pitfalls in the language design and thereby contribute to a more precise and better understood language description.

L8. The characteristics of translator implementations will not be dictated by the language definition or standards.

The adoption of a common language is a commitment to the current state of the art for programming language design for some duration. It does not, however, prevent access to new software and hardware technology, new techniques and new management strategies which do not impact the source language definition. In particular, innovation should be encouraged in the development of translators for a common language providing they implement exactly the source language as defined. Translators like all computer programs should be written in expectation of change.

L9. Translators for the language will be written in their own source language.

There will be at least one implementation of the translator in its own language which does all parsing and compile-time checking and produces an output suitable for easy translation to specific object machines. If the language is well-defined and uniform in structure, a self-description will contribute to understanding of the language. The availability of the machine independent portion of a translator will make the full power of the language available to any object machine at the cost of producing an additional code generator (whose cost may be high) and it reduces the likelihood of incompatible implementations. Translators written in their own source language are automatically available on any of their object machines providing the object machine has sufficient resources to support a compiler.

M. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only
2. Unambiguous Definition
3. Language Documentation Required
4. Control Agent Required
5. Support Agent Required
6. Library Standards and Support Required

M1. The language will be composed from features which are within the state of the art and any design or redesign which is necessary to achieve the needed characteristics will be conducted as an engineering design effort and not as a research project.

The adoption of a common language can be successful only if it makes available a modern programming language compatible with the latest software technology and is compatible with "best" current programming practice but the design and implementation of the language should not require additional research or require use of untried ideas. State-of-the-art cannot, however, be taken to mean that a feature has been incorporated in an operational DoD language and used for an extended period, or DoD will be forever tied to the technology of FORTRAN-like languages; but there must be some assurances through analysis and use that its benefits and deficiencies are known. The larger and more complex the structure, the more analysis and use that should be required. Language design should parallel other engineering design efforts in that it is a task of consolidation and not innovation. The language designer should be familiar with the many choices in semantic and syntactic features of language and should strive to compose the best of these into a consistent structure congruous with the needed characteristics. The language should be composed from known semantic features and familiar notations, but the use of proven feature should not necessarily impose that notation. The language must not just be a combination of existing features which satisfy the individual requirements but must be held together by a consistent and uniform structure which acts to minimize the number of concepts, consolidates divergent features and simplifies the whole.

M2. The semantics of the language will be defined unambiguously and clearly. To the extent a formal definition assists in attaining these objectives, the language's semantics will be specified formally.

A complete and unambiguous definition of a common language is essential. Otherwise each translator will resolve the ambiguities and fill in the gaps in its own

unique way. There are currently a variety of methods for formal specification of programming language semantics but it remains a major effort to produce a rigorous formal description and the resulting products are of questionable practical value. The real value in attempting a formal definition is that it reveals incomplete and ambiguous specifications. An attempt will be made to provide a formal definition of any language selected but success in that effort should not be requisite to its selection. Formal specification of the language might take the form of an axiomatic definition, use of the Vienna Definition Language, or use of some other formal semantic system.

M3. The user documentation of the language will be complete and will include both a tutorial introductory description and a formal in-depth description. The language will be defined as if it were the machine level language of an abstract digital computer.

The language should be intuitively correct and easily learned and understood by its potential users. The language definition might include an Algol-60 like description(P. Naur (Ed.), "Revised Report on the Algorithmic Language Algol-60," Communication of the A.C.M. Vol.6, No. 1, January 1963, p. 1-17.) with the source language syntax given in BNF or some other easily understood metalanguage and the corresponding semantics given in English. As with the descriptions of digital computer hardware, the semantics and syntax of each feature must be defined precisely and unambiguously. The action of any legal program will be determinable from the program and the language description alone. Any computation which can be described in the language will ultimately draw only on capabilities which are built into the language. No characteristics of the source language will be dependent on the idiosyncrasies of its translators.

The language documentation will include syntax, semantics and examples of each language construct, listings of all key words and language defined defaults. Examples shall be included to show the intended use of language features and to illustrate proper use of the language. Particularly expensive and inexpensive constructs will be pointed out. Each document will identify its purpose and prerequisites for its use.

M4. The language will be configuration managed throughout its total life cycle and will be controlled at the DoD level to ensure that there is only one version of the source language and that all translators conform to that standard.

Without controls a hopefully common language may become another umbrella under which new languages proliferate while retaining the same name. All compilers will be tested and certified for conformity to the standard specification and freedom from known errors prior to their release for use in production projects. The language manager will be on the OSD staff, but a group within the Military Departments or Agencies might act as the executive agent. A configuration control board will be instituted with user representation and chaired by a member of the OSD staff.

M5. There will be identified support agent(s) responsible for maintaining the translators and for associated design, development, debugging and maintenance aids.

Language commonality is an essential step in achieving software commonality, but the real benefits accrue when projects and contractors can draw on existing software with assurance that it will be supported, when systems can build from off the shelf components or at least with common tools, and when efforts can be spent to expand existing capabilities rather than building from scratch. Support of common widely used tools and aids should be provided independent of projects if common software is to be widely used. Support should be on a DoD-wide basis with final responsibility resting with a stable group or groups of qualified in-house personnel.

M6. There will be standards and support agents for common libraries including application-oriented libraries.

In a given application of a programming language three levels of the system must be learned and used: the base language, the standard library definitions used in that application area, and the local application programs. Users are responsible for the local application programs and local definitions but not for the language and its libraries which are used by many projects and sites. A principal user might act as agent for an entire application area.

SAI-78-523-110-SF

5,3.1

# HIGH ORDER LANGUAGE EVALUATION PROJECT

Final Report

Contract No. F030602-76-C-0165

Prepared For:

Rome Air Development Center
Griffiss Air Force Base
Rome, New York 13441

By:

E. F. Miller, Jr.
A. I. Wasserman

February 28, 1977

SCIENCE APPLICATIONS, INCORPORATED

Software Technology Center
211 Sutter St., San Francisco, CA 94108
(415) 421-9111

# PREFACE

This Final Technical Report contains comments, evaluations, and recommendations made by the HOL Evaluation Project Team at Science Applications, Inc. (SAI), Software Technology Center (STC), from November 1976 through February 1977.

Four languages were evaluated: ALGOL 60, ALGOL 68, LIS, and CS-4. Evaluations against TINMAN requirements on a line-by-line basis, against TINMAN desires, against language design criteria, and against baseline language selection criteria were completed.

In addition, this report comments upon the TINMAN requirements and makes recommendations concerning the direction of future efforts by the HOLWG and the methodology to be used in a language design and evaluation effort. This material is predicated on the assumption that the present work will lead to such an effort. A portion of this material was presented orally to Dr. Sera Amoroso in a technical meeting held at STC in San Francisco, California, 16 December 1976.

TABLE OF CONTENTS


| | | Page |
|---|---|---|
| CHAPTER 1. | OVERVIEW AND CONCLUSIONS | 1-1 |
| Section 1. | Overview and Summary of HOL Evaluation Project Results | 1-2 |
| | 1. Introduction | 1-3 |
| | 2. HOL Evaluation Project Description | 1-3 |
| | 3. Detailed Language Analysis | 1-4 |
| | 4. Analysis by Major TINMAN Areas | 1-6 |
| | 5. General Language Design Criteria | 1-8 |
| | 6. Toward Design of a New Language | 1-9 |
| CHAPTER 2. | METHODOLOGY | 2-1 |
| Section 1. | Methodology for Evaluation of High Order Languages | 2-2 |
| | 1. Introduction | 2-3 |
| | 2. Detailed Analysis of Language Requirements | 2-3 |
| | 3. Evaluation Against TINMAN's Major Categories | 2-4 |
| | 4. Evaluation Against Language Design Criteria | 2-4 |
| | 5. Base Language Criteria | 2-5 |
| Section 2. | Overall Language Evaluation Tabulations | 2-7 |
| | 1. Introduction | 2-8 |
| | 2. Language Summaries | 2-8 |
| Section 3. | Criteria for Evaluation of High-Order Languages (HOLs) | 2-12 |
| | 1. Introduction | 2-13 |
| | 1.1 Abstraction | 2-13 |
| | 1.2 Modularity | 2-14 |
| | 1.3 Linear Flow of Control | 2-14 |
| | 1.4 Control of Scope and Binding | 2-15 |
| | 1.5 Readability and Comprehensibility | 2-15 |
| | 1.6 Procedures and Data | 2-15 |
| | 1.7 Language Size | 2-15 |
| | 1.8 Analyzability | 2-16 |
| Section 4. | Criteria for TINMAN "Base Language" Selection | 2-17 |
| | 1. Introductory Notes | 2-18 |
| | 2. Language Design and Definition | 2-18 |
| | 3. Compiler Specification and Design | 2-18 |
| | 4. Compiler Implementation | 2-19 |
| | 5. Language Control and Maintenance | 2-19 |
| | 6. Commentary | 2-19 |

TABLE OF CONTENTS (Cont.)


| | | Page |
|---|---|---|
| CHAPTER 3. | TINMAN | 3-1 |
| Section 1. | Comments on the TINMAN Requirements | 3-2 |
| | 1. Introduction | 3-3 |
| | 2. On High Level Languages | 3-4 |
| | 3. Abstraction vs. Efficiency in TINMAN | 3-5 |
| | 3.1 Compile Time Costs | 3-6 |
| | 3.2 Optimization | 3-6 |
| | 3.3 Language Extensions | 3-7 |
| | 3.4 Expression Evaluation | 3-8 |
| | 3.5 Direct Code | 3-8 |
| | 4. Hiding of Types | 3-9 |
| | 5. Numeric Arithmetic | 3-10 |
| | 6. Data Structures | 3-11 |
| | 7. The GO TO | 3-11 |
| | 8. Extension of Operators | 3-12 |
| | 9. Programmability | 3-12 |
| | 10. Analyzability | 3-13 |
| | 11. Miscellaneous Issues | 3-14 |
| | 12. Summary | 3-15 |
| CHAPTER 4. | ALGOL 60 | 4-1 |
| Section 1. | Evaluation of ALGOL 60 Against Language Design Criteria | 4-2 |
| | 1. Introduction | 4-3 |
| | 2. Abstraction | 4-3 |
| | 3. Modularity | 4-3 |
| | 4. Linear Flow of Control | 4-4 |
| | 5. Control of Scope and Binding | 4-5 |
| | 6. Readability and Comprehensibility | 4-5 |
| | 7. Procedures and Data | 4-6 |
| | 8. Language Size | 4-6 |
| | 9. Analyzability | 4-6 |
| | 10. Summary | 4-7 |
| Section 2. | Evaluation of ALGOL 60 Against the TINMAN | 4-8 |
| | 1. Overview | 4-9 |
| | 2. Data and Types | 4-9 |
| | 3. Operations | 4-10 |
| | 4. Expressions and Parameters | 4-11 |
| | 5. Variables, Literals, and Constants | 4-12 |
| | 6. Definition Facilities | 4-12 |
| | 7. Scopes and Libraries | 4-12 |
| | 8. Control Structures | 4-13 |

TABLE OF CONTENTS (Cont.)


| | | Page |
|---|---|---|
| Section 2. | Evaluation of ALGOL 60 Against the TINMAN (Cont.) | |
| | 9. Syntax and Comment Conventions..... | 4-14 |
| | 10. Defaults, Conditional Compilation, and Language Restrictions..... | 4-15 |
| | 11. Efficient Object Representations and Machine Dependencies..... | 4-15 |
| | 12. Program Environment..... | 4-16 |
| | 13. Translators..... | 4-17 |
| | 14. Language Definition, Standards, and Control..... | 4-17 |
| Section 3. | Language Evaluation Summary for ALGOL 60..... | 4-19 |
| Section 4. | Detailed Comparison of ALGOL 60 to the TINMAN. | 4-21 |
| | 1. Introduction..... | 4-22 |
| CHAPTER 5. | ALGOL 68..... | 5-1 |
| Section 1. | Evaluation of ALGOL 68 Against Language Design Criteria..... | 5-2 |
| | 1. Introduction..... | 5-3 |
| | 2. Abstraction..... | 5-3 |
| | 3. Modularity..... | 5-4 |
| | 4. Linear Flow of Control..... | 5-4 |
| | 5. Control of Scope and Binding..... | 5-5 |
| | 6. Readability and Comprehensibility..... | 5-5 |
| | 7. Procedures and Data..... | 5-6 |
| | 8. Language Size..... | 5-6 |
| | 9. Analyzability..... | 5-7 |
| | 10. Summary..... | 5-7 |
| Section 2. | Evaluation of ALGOL 68 Against the TINMAN.... | 5-8 |
| | 1. Overview..... | 5-9 |
| | 2. Data and Types..... | 5-10 |
| | 3. Operations..... | 5-11 |
| | 4. Expressions and Parameters..... | 5-14 |
| | 5. Variables, Literals, and Constants..... | 5-17 |
| | 6. Definition Facilities..... | 5-18 |
| | 7. Scopes and Libraries..... | 5-19 |
| | 8. Control Structures..... | 5-20 |
| | 9. Syntax and Comment Conventions..... | 5-21 |
| | 10. Defaults, Conditional Compilation, and Language Restrictions..... | 5-23 |
| | 11. Efficient Object Representation and Machine Dependencies..... | 5-24 |

TABLE OF CONTENTS (Cont.)


| | | Page |
|---|---|---|
| Section 2. | Evaluation of ALGOL 68 Against the TINMAN (Cont.) | |
| | 12. Program Environment | 5-24 |
| | 13. Translators | 5-24 |
| | 14. Language Standards, Definition, and Control | 5-25 |
| Section 3. | Language Evaluation Summary for ALGOL 68 | 5-27 |
| Section 4. | Detailed Comparison of ALGOL 68 to the TINMAN | 5-29 |
| | 1. Introduction | 5-30 |
| CHAPTER 6. | LIS | 6-1 |
| Section 1. | Evaluation of LIS Against Language Design Criteria | 6-2 |
| | 1. General Comments | 6-3 |
| | 2. Abstraction | 6-4 |
| | 3. Modularity | 6-5 |
| | 4. Linear Flow of Control | 6-5 |
| | 5. Control fo Scope and Binding | 6-6 |
| | 6. Readability and Comprehensibility | 6-6 |
| | 7. Procedures and Data | 6-7 |
| | 8. Language Size | 6-7 |
| | 9. Analyzability | 6-7 |
| Section 2. | Evaluation of LIS Against the TINMAN | 6-9 |
| | 1. Overview | 6-10 |
| | 2. Data Types | 6-11 |
| | 3. Operations | 6-12 |
| | 4. Expressions and Parameters | 6-13 |
| | 5. Variables, Literals, and Constants | 6-13 |
| | 6. Definition Facilities | 6-14 |
| | 7. Scopes and Libraries | 6-15 |
| | 8. Control Structures | 6-15 |
| | 9. Syntax and Comment Conventions | 6-16 |
| | 10. Defaults, Conditional Compilation, and Restrictions | 6-18 |
| | 11. Efficient Object Representations and Machine Dependencies | 6-18 |
| | 12. Program Environment | 6-19 |
| | 13. Translators | 6-19 |
| | 14. Language Definition, Standards and Control | 6-20 |

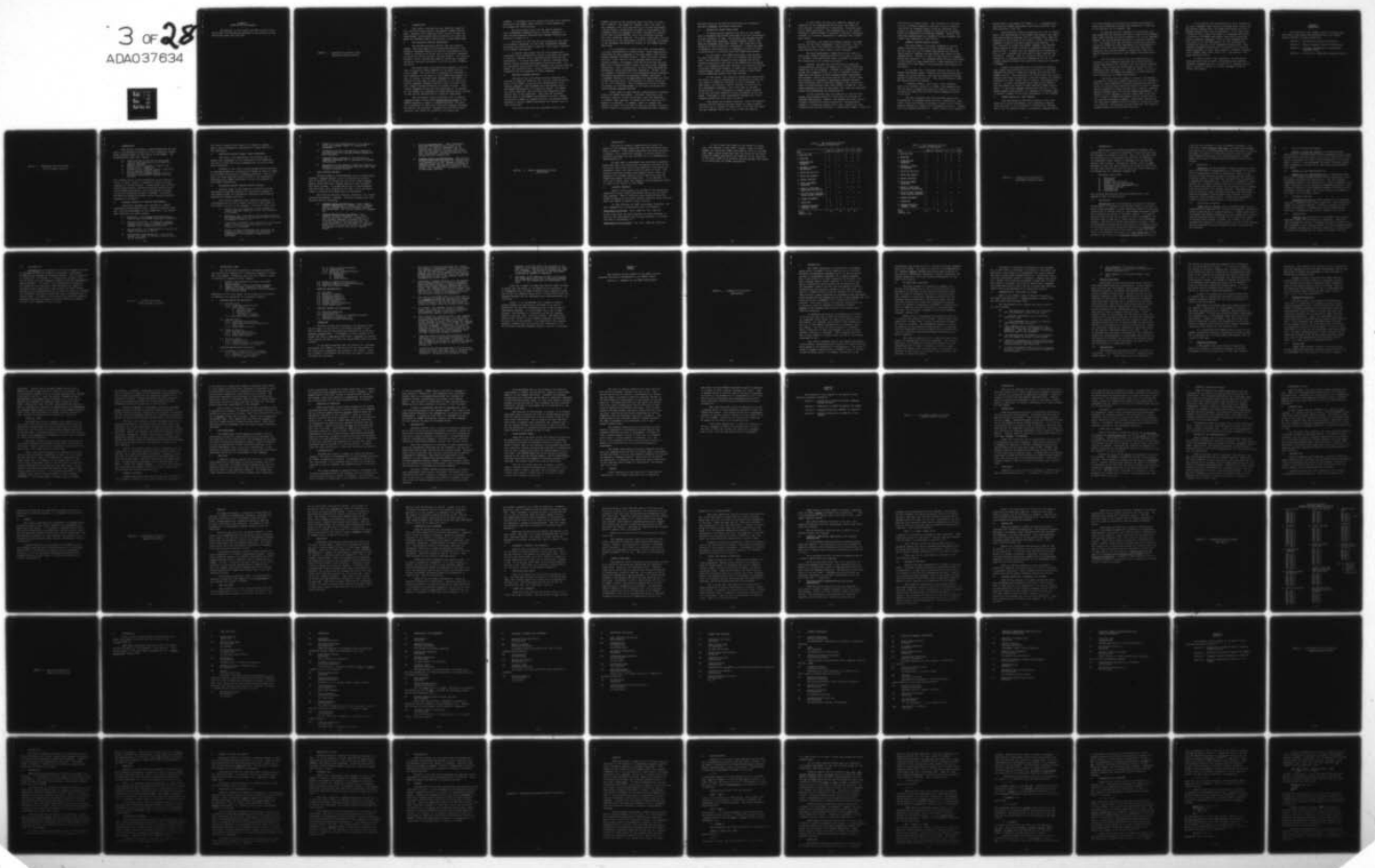
TABLE OF CONTENTS (Cont.)


| | | Page |
|---|---|---|
| Section 3. | Language Evaluation Summary for LIS..... | 6-21 |
| Section 4. | Detailed Comparison of LIS to the TINMAN..... | 6-23 |
| | 1. Introduction..... | 6-24 |
| CHAPTER 7 | CS-4..... | 7-1 |
| Section 1. | Evaluation of CS-4 Against Language Design Criteria..... | 7-2 |
| | 1. Introduction..... | 7-3 |
| | 2. Abstraction..... | 7-3 |
| | 3. Modularity..... | 7-4 |
| | 4. Linear Flow of Control..... | 7-5 |
| | 5. Control of Scope and Binding..... | 7-6 |
| | 6. Readability and Comprehensibility..... | 7-7 |
| | 7. Procedures and Data..... | 7-8 |
| | 8. Language Size..... | 7-8 |
| | 9. Analyzability..... | 7-9 |
| | 10. Summary..... | 7-9 |
| Section 2. | Evaluation of CS-4 Against the TINMAN..... | 7-10 |
| | 1. Overview..... | 7-11 |
| | 2. Data and Types..... | 7-12 |
| | 3. Operations..... | 7-13 |
| | 4. Expressions and Parameters..... | 7-14 |
| | 5. Variables, Literals, and Constants..... | 7-16 |
| | 6. Definition Facilities..... | 7-16 |
| | 7. Scopes and Libraries..... | 7-17 |
| | 8. Control Structures..... | 7-18 |
| | 9. Syntax and Comment Conventions..... | 7-19 |
| | 10. Defaults, Conditional Compilation, and Language Restrictions..... | 7-20 |
| | 11. Efficient Object Representations and Machine Dependencies..... | 7-20 |
| | 12. Program Environment..... | 7-21 |
| | 13. Translators..... | 7-22 |
| | 14. Language Definition, Standards, and Control..... | 7-22 |
| Section 3. | Language Evaluation Summary for CS-4..... | 7-24 |
| Section 4. | Detailed Comparison of CS-4 to TINMAN..... | 7-26 |
| | 1. Introduction..... | 7-27 |
| CHAPTER 8. | REFERENCES..... | 8-1 |

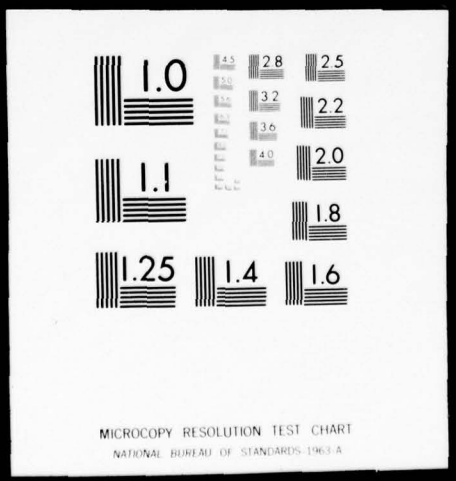
AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

3 OF 28
ADA037634



3 OF 28
ADA037634



# CHAPTER 1
# OVERVIEW AND CONCLUSIONS

The material in this chapter provides overview infor
mation about the methodology employed in this project and t
conclusions drawn from the work.

# SECTION 1. OVERVIEW AND SUMMARY OF HOL EVALUATION PROJECT RESULTS

## 1. INTRODUCTION

This report evaluates how well ALGOL 60, ALGOL 68, LIS, and CS-4 satisfy specific and general TINMAN requirements, assesses their compliance with criteria of good language design, and analyzes their suitability as a base language for extension and/or modification to meet HOLWG needs.

## 2. HOL EVALUATION PROJECT DESCRIPTION

The SAI HOL Evaluation Project had four distinct phases. The first phase involved detailed analysis of the candidate languages against the specific requirements of the TINMAN to determine the extent each of the subject languages satisfied (or did not satisfy) the requirements. The analyses produced "scores" showing the degree to which each language satisfied, partially satisfied, or did not satisfy, TINMAN's specifics.

The second phase considered each language in the context of evaluation areas suggested by the major sections within TINMAN. The intent was to evaluate the conformity of each language with the spirit of the TINMAN requirements, in hope of finding a suitable base language for the TINMAN, or finding one that might be easily modified to fit that role. Certain languages addressed specific areas of concern quite well, despite having relatively low overall "scores." Also, certain languages follow the ideas expressed in (or implied by) the TINMAN, but possessed or lacked features which resulted in only partial satisfaction of a specific requirement.

The third phase involved drafting a set of general language design criteria which abstracted the intent of the TINMAN regarding a number of language features. The goal was to determine if there was a good base language candidate, and whether the individual languages had particular characteristics that should be included in an eventual DoD

language. A secondary activity during this phase was a detailed analysis of the TINMAN, which resulted in some suggestions for changes and clarifications.

The fourth phase dealt with the need to design a "new" programming language using an existing language as a base and simultaneously examined some language features that should be included in it.

One result of this work was the recognition that none of the subject languages, nor any other programming language with which the Project Team is familiar, fully meets a majority of the TINMAN requirements.

It was for this reason that the later phases were undertaken, which addressed the issues of modifying the TINMAN to specify a feasible set of language design objectives and requirements. An effort was made to identify languages which handled certain extended requirements particularly well. It is anticipated that suitable base languages will include ALGOL 68, PASCAL, and (possibly) PL/I, so that selected features can be integrated into the framework of those langauges.

## 3. DETAILED LANGUAGE ANALYSIS

The work in the first phase involved comparing the features of ALGOL 60, ALGOL 68, LIS, and CS-4 with the requirements of the TINMAN, using available documentation on the various languages. Each language was rated against each of the TINMAN requirements in an objective manner. Following the initial analyses, each evaluation was repeated to make certain that it was consistent. This step was necessary because some of the TINMAN's requirements are quite complex and involve a number of language features, and because understanding of certain TINMAN requirements only became clear on this second pass.

The results show that the languages satisfied the

TINMAN criteria in the following order (from best to worst): CS-4, ALGOL 68, LIS, ALGOL 60. However, CS-4, the "best" of the languages, only fully satisfied 42 of the 98 requirements. A second measure, determining the number of requirements that languages completely failed to satisfy (intended as a measure of the ease of modification) led to the following ordering: LIS, CS-4, ALGOL 68, ALGOL 60. CS-4 again ranked high, failing to satisfy 22 of the requirements; even so, CS-4 was the weakest of the languages considered for several of the TINMAN requirement areas.

It is necessary to caution the reader against making a literal interpretation of these scores. Other organizations evaluating the same languages in the same manner would likely produce somewhat different results. There are several reasons for this. First, not all of the language definitions are entirely precise; for example, both the CS-4 and the LIS documents are "reference manuals" rather than more formal language definitions. CS-4, in particular, is still being designed and consequently leaves many questions unanswered. Second, there is a degree of subjectivity in distinguishing between completely satisfying and partially satisfying a TINMAN requirement. If a language conforms to the idea behind the requirement and fails to meet one minor point, it is possible to justify either an "S" or a "P" evaluation. This is especially true if a feature is missing, but the requirement is very easy to achieve with other existing language features.

Third, there is some room for interpretation of meaning within the TINMAN requirement. Different individuals have different ideas as to what constitutes "very few" keywords or a "minimal" base language, or whether safe and effective optimization is possible. In summary, there is a danger that subjective issues could be used to make one language look particularly viable as a candidate while downgrading other languages.

The self review of the detailed evaluations was intended to guarantee internal consistency among them.

## 4. ANALYSIS BY MAJOR TINMAN AREAS

It became quickly apparent from study of the TINMAN that no language exists which satisfied all of the requirements, or even a majority of them. The analysis was extended informally to some other languages, including PASCAL, Euclid, and some new languages still under design. This task led to confirmation of this conclusion. Accordingly, a search was begun to determine the suitability of languages to serve as a base of modification to ultimately meet the TINMAN requirements.

In this phase, the TINMAN requirements were grouped by areas (Data Types, etc.) and, at a somewhat less detailed level, the conformity between a given language and the key concepts contained in that area of the TINMAN were considered. Thus, if a language might have received a low score for failing to provide fixed point numbers, it might still rate high for being strongly typed, for providing a variety of data types, and for supporting compile-time type checking.

At this stage, the evaluation became less quantifiable. The result is a set of impressions of the extent to which certain ideas in TINMAN are captured in a given language. To some degree, one obtains a better "feeling" for the utility of a programming language at this level than at the more detailed level. It is necessary to examine sample programs to see how one might construct certain statements in the language. Knowledge of other programming languages and of programming methodology also comes into play.

This phase was extremely important, since the numerical scoring from the detailed analysis phase was indeed somewhat arbitrary and not especially useful once it had been concluded that no languages really came close to what was desired.

At this stage, one sees that ALGOL 68, despite the complexity of the language definition and a few complications related to elaboration of expressions, has great power and is highly consistent. The orthogonality of the various features of ALGOL 68 makes it possible to add new (orthogonal) features, or to delete unnecessary or unwanted ones, without major impact.

Similarly, LIS offers two elegant concepts: the PLEX notion, and the "data independent" type definition. The LIS approach to lower level features is also quite appealing, being handled more cleanly than in other languages.

The gross weaknesses of ALGOL 60 are also clear at this stage, thereby relegating ALGOL 60 to a place of historical interest (with significant impact on PASCAL and ALGOL 68) but eliminating it as a serious contender.

Finally, one sees that CS-4 has serious flaws and inconsistencies caused by the absence of a strong theoretical foundation for the language. Although this judgment may seem strong, CS-4 suggests itself as a set of language features added onto one another as more language design requirements were imposed, rather than having derived from a careful study and synthesis of fundamental programming language concepts. These weaknesses seem to make CS-4 somewhat less suitable as a basis for modifications than either PASCAL or ALGOL 68. CS-4 seems comparable to PL/1 in this sense, but lacks the experience gained from programming and implementation.

It was concluded that only ALGOL 68 (of the four languages being studied carefully) could serve as the base for language modification. However, the changes that would have to be made to ALGOL 68 are quite substantial, including major changes to incorporate abstract data types, enumeration types, exception and interrupt handling. A significant number of deletions

would have to be made as well. The conclusion is that adoption of a standard language by DoD would involve a substantial effort in language design, either by beginning from scratch or (more likely) making an extensive set of changes within the framework of an existing language. However, the resulting language might not resemble the base language very closely upon completion of that effort.

## 5. GENERAL LANGUAGE DESIGN CRITERIA

The third phase of the study attempted to use the TINMAN criteria as a basis for direct language design. It was felt that the list of TINMAN requirements was directed toward specific language features rather than toward broader language goals. Further, it was felt that goals for language design were important. The language design criteria established in this phase bore some resemblance to those presented in the introduction to the TINMAN, but had somewhat less emphasis on efficiency.

At the same time, a different view was taken of the TINMAN requirements proper. Whereas they had previously been treated as a given constant, they were now regarded as being subject to modification as well. (The recent release of IRONMAN substantiates this premise.)

A separate chapter was written which addressed some of the internal conflicts in TINMAN, with emphasis upon the difficulty of supporting high-level abstract notions on one hand and supporting low-level machine dependent notions on the other.

The four languages were then evaluated against the following set of language design criteria: abstraction, modularity, scope and binding of variables, linear flow of control, program readability, separation of procedures and data, language size, and program analyzability. These criteria were thought to

capture much of the essence of TINMAN, i.e., a language which satisfied the TINMAN requirements would rate relatively high when evaluated in this way.

This evaluation stage was instructive in that it showed some of the shortcomings of the TINMAN requirements, and identified some potential problems with any language which might be developed as the outcome of this effort. First, the resulting language is likely to be quite large by most measures. It will certainly be larger than PASCAL, FORTRAN, and ALGOL 60, simply because it tries to cover an extremely broad set of applications. It seems that the language may be comparable to PL/1 in size, although one would hope for more coherence and uniformity. However, there is a serious question as to the feasibility of implementing this language on any but the largest computers (recall that subsets are explicitly forbidden in TINMAN).

Second, the requirement for an interface to machine code, despite any protection that may be built in, seriously degrades the degree to which any language could satisfy the TINMAN criteria. The availability of "direct code" has a major impact on the areas of modularity, scope and binding of variables, procedures vs. data, linear flow of control, and program readability. In short, any program which makes use of machine code can subvert the intent of many important TINMAN requirements. If the final language contains the facility to incorporate machine code directly, the language will probably fail to satisfy a number of the other requirements of the language.

## 6. TOWARD DESIGN OF A NEW LANGUAGE

The last phase was undertaken because it was clear there was a need to synthesize a new programming language using features, as far as possible, from existing languages. One of the oft-repeated credos of programming language design

is to use language features which are known and understood, rather than attempting to design and incorporate a large set of new features into a language. [19]

In studying the four languages, it was apparent that some objectives are achieved very well in some of the languages, while other features are poorly handled in all of the languages. However, the scoring method used fails to show this distinction. For example the type extension and operator extension features of ALGOL 68 closely approximate the requirements of TINMAN, while the other languages do not even come close. Similarly, CS-4 and LIS attempt to address the lower level machine dependent interfaces, while ALGOL 60 and ALGOL 68 do not.

The last major activity was the identification of features in the languages evaluated which should be given strong consideration for synthesis into the eventual language design. This activity was predicated on the choice of an ALGOL-like base language (i.e., ALGOL 68, PASCAL, or PL/1) to be modified for the TINMAN requirements (or, more precisely, for a revised set of TINMAN requirements).

The effort did not attempt to create a new language, although the conclusion reached indicated this to be necessary. There is a very substantial difference between the identification of features which should be suitable for such a language, and the synthesis of those features into a coherent, formally defined language which fulfills the TINMAN requirements. The design of a new language is further complicated by the fact that a number of TINMAN requirements are not handled well in any existing programming language. Among these requirements are fixed point numbers and the avoidance of truncation, exception handling, asynchronous interrupt handling, and tradeoffs between compilation and runtime efficiency.

It is expected that extrapolation of these findings to the other languages under study by the HOLWG would confirm that no existing language meets the TINMAN requirements and that a language design effort must be undertaken. It is likely that a directed effort toward the design of a language which meets a revised set of TINMAN requirements would end up producing several candidate languages. None of these languages would fully meet all of the requirements, but all would be more satisfactory than any of the languages evaluated here. Such candidate languages would be able to draw not only on established languages, but also upon a number of research languages (CLU, ALPHARD, MESA, GYPSY, PLAIN, etc.) and proposed language features as well.

If such an effort were undertaken, the methodology developed here would be useful in analyzing newly designed candidate languages against a revised set of language requirements, eventually leading to a single language having DoD control and support.

# CHAPTER 2
# METHODOLOGY

The sections in this chapter relate to criteria that were used in the evaluations performed and to the overall results obtained in those evaluations.

Section 1. HOL Evaluation Project Methodology

Section 2. Overall Language Evaluation Tabulations

Section 3. Criteria for Evaluation of High-Order Languages (HOLs)

Section 4. Criteria for TINMAN Base Language Selection

# SECTION 1. METHODOLOGY FOR EVALUATION OF HIGH ORDER LANGUAGES

## 1. INTRODUCTION

This section describes a staged methodology that was used in the systematic evaluation of the properties of High Order Languages (HOLs) as they relate to the TINMAN. The methodological steps were, briefly:

- Identification of the role of the Tinman
- Item by item comparison of the Tinman to a specific language
- Evaluation of a language in terms of the Tinman's major categories
- Evaluation of a language against a specified set of language design criteria
- Evaluation of a language against a set of carefully designed base language selection criteria

The intended result of this methodology is the selection of one, or more, candidate languages to serve as a base language for further studies. Accomplishing this evidently implies simultaneously assessing potential modifications of the TINMAN requirements, and investigating the possibilities for extending candidate languages. Ultimately the modified TINMAN requirements and an extended candidate language will converge.

## 2. DETAILED ANALYSIS OF LANGUAGE REQUIREMENTS

Individual languages were compared on a TINMAN requirement-by-requirement basis. The basic question asked was: "Are the requirements of the TINMAN met?" The answers were organized into these classes:

- Satisfied - the language being analyzed was found to meet the TINMAN requirement completely.
- Partially Satisfied - at least one language property did not meet the TINMAN requirement, although other language properties did.
- Not Satisfied - the language did not satisfy the TINMAN requirement in any way.
- Undetermined (Undeterminable) - for various reasons the answer to the basic question could not be determined.

This latter category would apply for a number of reasons, mostly lack of documentary information or inapplicability of the requirement.

3. EVALUATION AGAINST TINMAN'S MAJOR CATEGORIES

This step in the methodology was concerned with a higher level assessment of the intentions of the TINMAN and the level to which particular languages met (or did not meet) those intentions.

Analyses at this level involved generalized discussions of language properties, including such issues as style of organization and/or structuring of language properties. Much of the basis for these analyses existed already in the discussions supporting each TINMAN requirement, as well as elsewhere in the TINMAN documentation.

4. EVALUATION AGAINST LANGUAGE DESIGN CRITERIA

The next stage in the sequence of analyses involved addressing the general goodness of a particular language in very general terms, not necessarily implying either match or mismatch with TINMAN objectives.

The criteria themselves were chosen as categories of investigation and critical comment that have a high impact on the ultimate utility of the language. The topics chosen were:

1. Abstraction--the capability of the language to cleanly represent separable "parts" of a solution to a problem.

2. Modularity--the contribution the language makes to the understandability and production ease (by humans) of programs written in it.

3. Linear Flow of Control--the contribution the language makes to ease with which the control is self-evident to a programmer.

4. Control of Scope and Binding--the facilities the language provides to minimize the complexity of programming in the presence of complex data structures.

5. Readability and Comprehensibility--the degree to which the language "reads" to a proficient programmer.

6. Procedures and Data--the measure of separation provided within the language between program and data structure.

7. Language Size--a measure of the difficulty a competent programmer has in learning the language from scratch.

8. Analyzability--the extent to which the language is amenable to automated program analysis tools used to assure software quality.

## 5. BASE LANGUAGE CRITERIA

The next stage in the methodology was to evaluate each candidate language against a set of criteria that could be used to determine its suitability for use as a base language against which additional language definition and refinement work could proceed. In accomplishing this it was recognized from the outset that, potentially at least, all of the major stages of language development must be represented.

The criteria selected reflected, therefore, the "normal" process of developing a language. The major elements that were involved were the following:

- Language Design and Definition. This category includes assessments of language style, language syntax, language semantics (including descriptive mechanisms), and tutorial aspects of language use.

- Compiler Specification and Design. This category includes assessments of the difficulty of building a compiler (translator) for the candidate language, estimates the level of effort needed to develop a detailed compiler specification, an assessment of algorithms needed within the compiler as they are implied by language features, and overall compiler design.

- Compiler Implementation. These assessments deal with the overall costs likely to be incurred in a compiler implementation (or compiler modification) activity. They deal also with the impact of requirements for self-boot, optimization, interactive operation, and program analysis capabilities.

- Language Control and Maintenance. This area can be partitioned into sub-areas that address existing dissemination of the language (assuming the language already exists), potential problems of controllability, and compiler validation techniques. Also included are assessments of control of subsidiary features.

# SECTION 2. OVERALL LANGUAGE EVALUATION TABULATIONS

## 1. INTRODUCTION

This section presents summary material about the results of the language evaluations that are reported in more detail elsewhere. Each TINMAN requirement was rated according to whether, for a particular language, it was satisfied (S), partially satisfied (P), not satisfied (N), or undetermined or undeterminable (U).

In some cases a requirement was classified as undetermined when there was insufficient information available to the evaluation team at the time of the evaluation; this should not necessarily imply that such information does not exist. The partially-satisfied category was assigned when at least one instance of a language property did not explicitly meet the TINMAN requirement or requirement part. It is important to note that in these cases, the judgments were deliberately harsh in terms of the "spirit" of the TINMAN.

## 2. *LANGUAGE SUMMARIES*

Tables 1 and 2 present the outcome of the evaluations, summarized by major TINMAN requirement section and by individual language. Table 1 indicates those instances where the TINMAN is explicitly satisfied; Table 2 indicates those instances where the TINMAN is explicitly <u>not</u> satisfied.

In rank ordering for those requirements satisfied, the four languages are, in decreasing order of precedence:

<u>Requirements satisfied</u>: CS-4, ALGOL 68, LIS, ALGOL 60

On the other hand, the rank ordering for these languages in terms of the requirements that are <u>not</u> satisfied, in increasing order of precedence, is:

<u>Requirements not satisfied</u>: LIS, CS-4, ALGOL 68, ALGOL 60.

It can be seen that based on this method of evaluation, CS-4 appears to rate "best" overall, since it scores highest for satisfying TINMAN requirements, while at the same time comes a close second best when the scores for not satisfying the TINMAN requirements are compared. On the other end of the scale, ALGOL 60 is farthest from satisfying, and closest to not satisfying the TINMAN requirements.

Table 1. HOL Evaluation Project
Overall S-Comparisons

| Topic | ALGOL 60 | ALGOL 68 | LIS | CS-4 | Weight |
|---|---|---|---|---|---|
| A. DATA AND TYPES | 1 | 1 | 2 | 3 | 16 |
| B. OPERATIONS | 2 | 5 | 5 | 4 | 8 |
| C. EXPRESSIONS AND PARAMETERS | 3 | 1 | 4 | 3 | 11 |
| D. VARIABLES, LITERALS, AND CONSTANTS | 0 | 4 | 5 | 4 | 7 |
| E. DEFINITION FACILITIES | 2 | 5 | 2 | 5 | 14 |
| F. SCOPES AND LIBRARIES | 2 | 3 | 2 | 3 | 8 |
| G. CONTROL STRUCTURES | 0 | 1 | 0 | 2 | 11 |
| H. SYNTAX AND COMMENT CONVENTIONS | 5 | 4 | 6 | 5 | 8 |
| I. DEFAULTS, CONDITIONAL COMPILATION, RESTRICTIONS | 3 | 2 | 3 | 4 | 8 |
| J. EFFICIENT OBJECT REPRESENTATION, MACHINE DEPENDENCE | 1 | 0 | 1 | 3 | 9 |
| K. PROGRAM ENVIRONMENT | 1 | 2 | 3 | 1 | - |
| L. TRANSLATORS | 0 | 2 | 3 | 4 | - |
| M. LANGUAGE DEFINITION, STANDARDS, CONTROL | 2 | 3 | 1 | 1 | - |
| TOTALS (Maximum = 98) | 22 | 34 | 37 | 42 | |

Table 2. HOL Evaluation Project
Overall N-Comparisons

| Topic | ALGOL 60 | ALGOL 68 | LIS | CS-4 | Weight |
|---|---|---|---|---|---|
| A. DATA AND TYPES | 5 | 1 | 4 | 1 | 16 |
| B. OPERATIONS | 5 | 2 | 4 | 2 | 8 |
| C. EXPRESSIONS AND PARAMETERS | 3 | 3 | 3 | 5 | 11 |
| D. VARIABLES, LITERALS, AND CONSTANTS | 4 | 1 | 1 | 1 | 7 |
| E. DEFINITION FACILITIES | 5 | 2 | 4 | 2 | 14 |
| F. SCOPES AND LIBRARIES | 2 | 1 | 0 | 1 | 8 |
| G. CONTROL STRUCTURES | 4 | 2 | 3 | 3 | 11 |
| H. SYNTAX AND COMMENT CONVENTIONS | 2 | 1 | 1 | 3 | 8 |
| I. DEFAULTS, CONDITIONAL COMPILATION, RESTRICTIONS | 3 | 2 | 1 | 1 | 8 |
| J. EFFICIENT OBJECT REPRESENTATION, MACHINE DEPENDENCE | 3 | 4 | 1 | 0 | 9 |
| K. PROGRAM ENVIRONMENT | 2 | 1 | 0 | 1 | - |
| L. TRANSLATORS | 2 | 3 | 0 | 0 | - |
| M. LANGUAGE DEFINITION, STANDARDS, CONTROL | 2 | 3 | 0 | 2 | - |
| TOTALS (Maximum = 98) | 42 | 26 | 20 | 22 | |

# SECTION 3. CRITERIA FOR EVALUATION OF HIGH-ORDER LANGUAGES (HOLs)

# 1. INTRODUCTION

Design goals for computer programming languages vary considerably, as a function of developer's tastes and the needs the language is intended to fulfill. Thus, it is unlikely that any particular set of evaluation criteria will be sufficient (or acceptable) when a number of languages will be compared against those criteria. Fortunately, however, there is some agreement on what are "good" features for an HOL to have; a set of eight such factors can form the basis for initial analyses of the suitability of a candidate language [8].

The eight factors are:

- Abstraction
- Modularity
- Linear Flow of Control
- Control of Scope and Binding
- Readability and Comprehensibility
- Procedures and Data
- Language Size
- Analyzability

The following sections give detailed explanations of the meaning of each of these factors.

## 1.1 Abstraction

Abstraction has been recognized as a means to develop a representation of concepts which relates closely to the application being programmed, to hide inessential details of the problem solution at various levels of the program development process, and to support the notion of "top-down" design. If a problem solution involves the use of stacks or trees, for example, one should be able to make use of those objects in the programming process. The ability to define these abstract objects, along with appropriate operations on the objects, is extremely valuable. If the objects and their associated operators are encapsulated so that the representation of the object is hidden from other parts of the program, the facility for data abstraction is comparable to the facility for procedural abstraction provided by

functions and procedures in many programming languages. Such a facility, termed abstract data types, provides the programmer with opportunity to refine program and data structures in parallel and makes it possible to create data objects within a program which resemble those used in the problem solution, thereby easing the process of transforming the problem solution into a program.

## 1.2 Modularity

Modularity has been observed to make an important contribution to the overall comprehensibility of programs, to the practice of programming by levels of abstraction, and to the production of large software systems by allowing various pieces of a software system to be effectively isolated from one another in a conceptually meaningful way. The ability to decompose a large problem into a number of smaller ones and to clearly delineate the interactions among the pieces in an important tool in gaining intellectual mastery over complex problems. Programming aids such as structure charts and HIPO charts have been developed to help the programmer identify modules and represent the total structure of the software system. A module can correspond to either a procedural abstraction or a data abstraction.

## 1.3 Linear Flow of Control

Linear flow of control within a program module has been recognized to be important for its value in debugging, verification and the understanding of program operation. Formal verification of programs and program readability are both simplified by controlling the branching within a program and by reducing the number of possible paths through a program. As a result, the use of unrestrained transfers of control (GOTOs) has fallen into disrepute and many alternative and more powerful control structures have been proposed.

## 1.4 Control of Scope and Binding

Control of scope and binding of variables has been identified as a technique which reduces programming errors caused by side effects and which serves as an aid to verification. Such control is necessary in order to achieve modularity on a formal sense. Without it, the programmer may easily circumvent restrictions concerning the proper use of input and output parameters for a module.

## 1.5 Readability and Comprehensibility

Readability and comprehensibility of programs has been seen to be a valuable program property which contributes to ease of program maintenance and modification. The use of opaque programming tricks or the construction of cryptic programs is no longer considered to be an acceptable programming practice. Many properties combine to yield readable programs; including the use of mnemonic variable names, the liberal insertion of comments, and linear flow of program control.

## 1.6 Procedures and Data

Procedures and data have been recognized to be separate entities of programs with distinct properties. Data objects may change their values dynamically, whereas a procedure should be static and immutable. Programs which modify themselves are difficult to comprehend and virtually impossible to verify.

## 1.7 Language Size

Language size as measured by the number of key words or language concepts has been seen to be important. Relatively small languages are easier to implement and make it possible for the programmer to gain complete mastery of the programming language. The language designer must carefully synthesize a limited number of features from among the vast number of potential choices.

## 1.8 Analyzability

Analyzability is measured by the level of support provided by language features that simplify the process of analyzing already-written programs by automatic methods. (This is basically a "recognition" as opposed to a "compilation" problem, since one can assume that the language's compiler provides comprehensive syntax analyses.) In a sense, this characteristic crosses all the boundaries outlined above, since analyzability cannot be achieved without significantly good "scores" on other factors. Included in this characteristic is the ease with which the language can incorporate semi-active assertions and/or instrumentation of various kinds. In addition, the simplicity of symbolic analysis/execution/evaluation processes that would operate on programs written in the language is also a factor.

# SECTION 4. CRITERIA FOR TINMAN "BASE LANGUAGE" SELECTION

1. INTRODUCTORY NOTES

The outline below represents a proposed statement of the criteria for evaluation of a candidate language to "represent" the TINMAN. Ultimately there are four possible routes to choosing an appropriate base language:

1. Design a new language that meets the TINMAN specifications
2. Change TINMAN to match an existing language.
3. Change an existing language to meet TINMAN.
4. Compromise between an extended existing language and a reduced TINMAN.

Regardless of the route taken, the following set of criteria is essential for evaluation of a candidate language.

2. LANGUAGE DESIGN AND DEFINITION

- 2.1 Language Style
  - 2.1.1 Programmer acceptability
  - 2.1.2 Major Issues:
    - a. uniform referents
    - b. abstract types
    - c. extensibility
    - d. "nearness" to TINMAN
  - 2.1.3 Secondary Issues (Details)
- 2.2 Language Syntax
  - 2.2.1 Programmer acceptability
  - 2.2.2 Syntax Modification/Extension Cost, Time, Risk
  - 2.2.3 Psychological Factors
- 2.3 Language Semantics
  - 2.3.1 Informal description
  - 2.3.2 Semi-formal Description
  - 2.3.3 Formal Description
- 2.4 Tutorial Aspects
  - 2.4.1 Learning Curve
  - 2.4.2 Unlearning Curve (If applicable)
  - 2.4.3 Tutorial Subset--its clarity

3. COMPILER SPECIFICATION AND DESIGN

- 3.1 Assessment of Compilability of Language
  - 3.1.1 Modification of existing compiler
  - 3.1.2 Generation of new compiler
  - 3.1.3 Run-time support needed

3.1.4 Size of Syntax Processed
3.1.5 Diagnosibility
3.1.6 Efficient Code/Optimization
3.1.7 Reformatability
a. expansion
b. reference
c. other details

3.2 Design of Compiler Specification
3.3 Detailed Compilation Algorithm Analysis
3.4 High-level Compiler Design

4. COMPILER IMPLEMENTATION

4.1 Self-Boot
4.2 Production Version
4.3 Optimizing Capability
4.4 Debugging Capability
4.5 Interactive Capability
4.6 Library Capability
4.7 Program Analysis Capability
4.8 Other Capabilities

5. LANGUAGE CONTROL AND MAINTENANCE

5.1 Existing Dispersion
5.2 Controllability
5.3 Extension of Control to Subsidiary Features Application (libraries, etc.)
5.4 Validation Program Techniques

6. COMMENTARY

The above outline was produced on the assumption that the end result of the efforts of the HOLWG (perhaps in late 1977 or 1978) would be the design and definition of a new programming language DoD/1. It attempts to cover a number of the issues that must be addressed before such a language can successfully achieve the goals to which the entire HOLWG effort has been directed.

It has been recognized that the creation of a language which meets the TINMAN (or some future set of) requirements is not sufficient to guarantee the success of this effort. There are a number of concomitant factors which are equally important, including the following:

1. There must be effective mechanisms for teaching the language to programmers. The cost of instruction and the preparation of tutorial materials, combined with the loss of productivity while programmers achieve proficiency with a new language, is comparable in cost to the language design and implementation effort itself.

2. Accordingly, language features in a new language should resemble features in existing languages when those features serve a similar purpose and should differ when different purposes are to be served. The language should not be burdened with complicated or inconsistent rules of syntax, since they will be a source of programming errors and will be easily forgotten by programmers.

3. The value of the HOLWG effort is in the creation of a standard programming language which supports programming discipline and the creation of high quality software. Any activity which tends to dilute the standard must be strongly discouraged.

4. As a result, the language should be complete enough that there will be no need to extend the base language, except to develop libraries of procedures and abstract data types.

5. There should be a tutorial subset of the language, but no subset implementations. In other words, a programmer should not necessarily be taught the entire language initially, even though the language processors with which he/she will work will support the entire language. Furthermore, the tutorial materials developed for DoD/1 should present the language in the context of recent efforts in programming methodology and good programming style in order to motivate language conversion.

6. Compilers should be built with portability as an objective so as to encourage the availability of the language on a wide variety of machines. It may be valuable to develop an intermediate language such as OCODE used by BCPL in order to simplify portability.

7. Validation and testing mechanisms will have to be created for DoD/1 along the lines of the mechanisms which are in place for COBOL compiler testing validation. In addition to validating correct

programs, attention should also be given to the behavior of the compiler in the presence of incorrect programs. The validation software will need to be extremely rigorous due to the requirement concerning no subsets or supersets.

8. The HOLWG, an X3 committee of ANSI, or an equivalent organization will have to maintain responsibility for the language and its development over time.

There are a number of steps that must be taken in order to assure that the effort of the HOLWG will be successful. It is recommended that the HOLWG include a major evaluation component within the language design effort to make certain that the language designed and the implementations constructed meet the objectives.

Lastly, it is recommended that a number of generic program examples be specified by the HOLWG to be used to demonstrate the ease and comprehensibility of each candidate programming language. It is envisages that a set of small problems, each designed to illustrate the way the language handles a critical issue, would be created. The HOLWG would require code, written in the candidate language representing the solution to each problem, to be submitted by the language designer, thereby facilitating practical comparison of the way different proposed languages handled a variety of problems.

# CHAPTER 3
# TINMAN

The contents of this chapter of the report involve material relating to assessments of the TINMAN itself.

## Section 1. Comments on the TINMAN Requirements

# SECTION 1. COMMENTS ON THE TINMAN REQUIREMENTS

## 1. INTRODUCTION

The TINMAN represents an idealized set of language design goals, intended to bring together all of the recent advances in programming languages and all of the intriguing concepts in programming languages into a single language. What is envisioned is a single language (the universal computer language again?) which supports all of the key concepts of abstraction and modularity on one hand and gives the programmer control over the low-level details of object representation and execution regimes on the other hand. In addition, TINMAN envisions creating language control mechanisms by which a single organization (presumably some DoD agency) would oversee both the design and control of the language and the construction and management of all translators for all machines. This would involve an elaborate validation effort to ascertain that the implemented language conformed exactly to the language specification, with neither extensions nor omissions.

In examining the specific requirements more closely, it can be seen that some of them are unachievable in practical languages, while others are simply self-contradictory. Furthermore, the language which results from attempting to meet all of the TINMAN requirements is of necessity large and complex. The result of such a design is a language which is difficult to learn and remember, as well as being somewhat difficult to implement. This language would present a large number of concepts which are new to the average programmer as well.

This section examines some of the TINMAN requirements in more detail, paying particular attention to those objectives which appear to be unrealistic or contradictory. The intent is either to suggest modifications to TINMAN for use in the next evaluation/design phase or to suggest relaxations in the

requirements that should be used to evaluate existing languages against the TINMAN. The result of this set of recommendations is a somewhat smaller language which gives more attention to the issues of abstraction, modularity, control structures, and program readability, with less attention being given to the lower level details.

## 2. ON HIGH LEVEL LANGUAGES

The basic concept of a high-level language is to provide the programmer with a set of facilities which are meaningful in the terms of the problem being solved rather than in terms of the machine on which it is being solved. Early research in high-level languages confirmed that programmers were more productive and produced higher quality code when working with a high-level programming language. The use of algebraic expressions in FORTRAN or record structures in COBOL was far more meaningful to the programmer than was a set of machine language (or even macro assembly language) instructions.

Recent work in programming methodology has gone on to validate that concept still further. Abstract data types remove still another level of object presentation throughout much of a program. The programming language is seen as an extension of the problem-solving process; it is desired that the transformation from a program design (problem solution) to a correctly functioning program be simplified as much as possible.

The value of abstraction is to hide the lower level details of implementation from the programmer. The value of modularity is to manage the intellectual complexity of the programming task, to make it feasible to divide a large project into pieces, and to permit parts of a program to be isolated in such a way as to minimize side effects and the problems of connecting modules.

The goals of efficiency have generally been regarded as, at least partly, secondary to the goals of abstraction and modularity, even though it is recognized that a poor compiler may lead to much slower program execution. Inefficient execution-time performance tends to eliminate a language from actual use. Rather than giving the programmer a great deal of power with respect to storage allocation or object code, languages have been restricted in such a way as to insure that efficient object code can be generated by an optimizing compiler.

## 3. ABSTRACTION VS. EFFICIENCY IN TINMAN

There is an inherent conflict between the goals of efficiency and abstraction. TINMAN proposes to achieve both, and some significant problems result.

The requirements which address efficiency directly are the following:

B6. "... The operations 'and' and 'or' on scalars will be evaluated in short circuit mode."

C4. "...constant expressions will be evaluated before run time."

I2. "... The programmer will be able to override these defaults when necessary."

J1. "The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs."

J2. "Any optimizations performed by the translator will not change the effect of the program."

J3. "The source language will provide encapsulated access to machine dependent hardware facilities including machine code insertions."

J4. "It will be possible within the source language to specify the object presentation of composite data structures..."

J5. "The programmer will be able to specify whether calls on a routine are to have open or closed implementations..."

L3. "The translator will minimize compile time costs..."

## 3.1 Compile Time Costs

Requirement L3, stressing minimization of compile-time costs, seems especially problematical. First, requirement L6 states that the compiler should perform as much checking as possible, including type checking and verification of semantic restrictions. Second, the size of the proposed language is such that construction of an efficient translator is nontrivial. Third, minimization of compile-time costs seems contradictory to objectives of execution-time efficiency, since many of those gains come as a result of optimizations made at compile-time, often requiring multiple passes over the intermediate code or initial object code in order to generate efficient code. If one wishes to meet J1, for example, all DO loops (or their equivalent) must be checked to see when the step and termination values are constant and when they are variable (unless the language forbids this use of variable number of iterations). Fourth, the availability of compile-time facilities, including conditional compilation, represent an additional burden on the compiler, increasing compile time costs. Fifth, it is expected that most uses of programs will be in operation rather than in development; it would appear that the goal of efficiency would dictate that the effort be directed toward the generation of efficient executable code rather than toward minimization of compilation costs.

## 3.2 Optimization

The optimization requirements present a similar set of problems. J2 is particularly difficult, especially when viewed against C1: "Side effects which are dependent on

the evaluation order among the arguments of an expression will be evaluated left-to-right." C1 restricts a significant number of potential optimizations, including the collateral elaboration notion of ALGOL 68 and the grouping by hierarchy of LIS, whereby the multiplication would be performed first in A + B*C. It is recognized that, particularly for floating point numbers, any change in ordering of evaluation of an expression can change the result, due to round-off errors in the least significant decimal places. A + B - C may easily yield a different result than A - C + B, particularly if this sum is performed repeatedly.

In short, the combination of J2 and L3 would seem to indicate that no optimization whatsoever should be performed on programs, despite the fact that achievement of J1 requires such optimization.

Requirement J4 presents additional complications for the compiler. The ability of the programmer to specify the object representation of a composite data structure is very unpleasant. If there were severe main memory requirements for an application, the programmer might very well pack the data objects tightly as possible. The object code that would be required to extract the appropriate bits for each item of a composite data structure would impose considerable overhead at both compilation and execution time.

It is recognized that TINMAN wishes to give the programmer the ability to make the time/space efficiency tradeoff in programs, but the other TINMAN requirements then run counter to that goal.

## 3.3 Language Extensions

A key question that was raised by Standish and others with respect to extensible languages was whether programmers are able and qualified to make these language

extensions. This question will certainly arise in conjunction with the language derived from these requirements, and is also applicable to whether the programmer is best qualified to choose object representations.

Perhaps a higher level directive should indicate available space, with the compiler or optimizer trying to fit a program into a space by changing object representations if the compiled object code is too large. There is the danger that the programmer will go to considerable effort to optimize space utilization when space is not an issue at all for a given application.

### 3.4 Expression Evaluation

The use of short circuit mode for evaluation of 'and' and 'or' operations is a potential source of errors should programmers include functions having side effects within the Boolean expression to be evaluated. More complicated object code is needed to achieve the short circuiting of execution than is required to simply evaluate the Boolean expression. Again, the requirement for minimized compilation is contradicted by another TINMAN requirement. It is recognized that programmers must be taught not to program in such a way that side effects are created in expressions which are to be evaluated. If this is done, then there is no need for the requirement about side effects (C1). In short, performing short circuit evaluation of expressions according to B6 and then permitting side effects (C1), even if the subexpressions are evaluated left-to-right, is a likely source of programming errors.

### 3.5 Direct Code

TINMAN seems somewhat schizoid on the distinction between supporting high-order language constructs and giving the programmer all of the power of the machine language

programmer. Perhaps this is because TINMAN does not really address the set of applications to which it is intended. If the envisioned language is to be a system implementation language (like BLISS, LIS, etc.), then the machine language and data representation requirements may be needed; if it is to be an applications language for commercial data processing (like COBOL), then all of these low level issues should be removed in favor of presenting a clear set of abstractions to the programmer. The language simply cannot be all things to all people--efficient object code, efficient space utilization, minimal compilation time, suitable for abstraction and modularity, and so on.

## 4. HIDING OF TYPES

The TINMAN envisions a language with a powerful set of data types with a facility for introducing new data types, new operations on those types which are encapsulated with the type definition, and the ability to extend base language operations to the new types. The requirements are very clear about the use of types in the language, with emphasis on clarity and consistency. Yet there are some problems.

One wants to know the type of the object with which one is dealing and should be extremely careful about the mixing of data types. However, some of the requirements tend to blur this distinction. This comes about in several ways.

First, the equality comparison is supposed to be applicable to any two objects, regardless of type. It is not clear why this should be. What is the meaning of comparing a real number with a set, for example, or an integer with a record? A more useful requirement would be that the equality comparison would be applicable to two objects of the same type or to two objects of united type (using the ALGOL 68 term), whose intersection types are not null. If one variable was MODE A = UNION (INTEGER, REAL) and the other was MODE B = UNION (INTEGER, CHARACTER), it would make sense to compare them as integers,

for example. In general, programmers should have a consistent use of variables of types so that meaningless comparisons do not occur, rather than permitting the comparison operator to be universally applied. Another meaningful extension would be to require the comparison operator to be defined for all new programmer-defined data types (as is done in CS-4).

Second, assignment of conformable arrays and transfers between records or arrays of identical logical structure is a useful shorthand notation, particularly for records, but obscures the fact that the assignment applies to a complex data structure, thereby detracting from program comprehensibility and readability.

The object of providing a uniform reference notation is difficult and elusive. On the one hand, one would like to be consistent, but on the other hand, one would like to know something about the type of the object on which an operation is being performed. PASCAL, for example, has different reference notions for arrays, procedures, records, and pointer references. It is immediately apparent as to which type of object is being used. Geschke and Mitchell [18] have examined the problem and concluded that, while one could achieve this objective, it may not be worth doing so.

Fourth, the optional specification of parameters on the formal side of a procedure declaration (CS) does not seem to be the best way to deal with generic procedures, especially since one wants to constrain the operations that are performed on objects of various types. Even PL/I has a more intelligent solution. One may specify a parameter to be a united type, e.g., MODE A = (REAL, INT, STRING, BOOLEAN, ...) so that the procedure call can accept any parameter which is of one of those types. This provides the facility desired by CS.

## 5. NUMERIC ARITHMETIC

TINMAN requirements specify both fixed and floating point real arithmetic, despite the fact that fixed-point arithmetic

is the source of a significant number of execution time errors. It would appear that fixed point arithmetic is desired to avoid the potential difficulties of roundoff error associated with floating point computation, when carried out in binary. Yet fixed point arithmetic has the danger of truncating the most significant digits in arithmetic calculations, of losing signficant digits on both the high order and low order end, and adds several additional requirements to the language.

Much of what is required in fixed point arithmetic could be handled with integers; the remainder could almost certainly be handled by floating point arithmetic where the calculation is carried out in decimal rather than in binary. Binary can be considered the default case. This removes one type from the language, as well as a common source of execution-time errors. This would again provide the programmer with the choice between efficiency and space.

## 6. DATA STRUCTURES

The TINMAN goes to some lengths to propose a set of data objects as built-in types which provide a firm basis for the definition of new types and for the implementation of recursive data structures as discussed by Hoare. Hoare's contention is that the defined set of data structures makes use of the pointer unnecessary, even if its use is restricted. If one believes in recursive data structures, then pointers and pointer mechanisms should not be a requirement in the language.

## 7. THE GO TO

A similar situation applies for the GO TO. The TINMAN requirements go to some length to provide a powerful set of control structures, including exception-handling and loop termination from any point within a loop, the two situations for which the GO TO is now used most commonly. Yet TINMAN then turns around and requires a GO TO statement after rendering it

totally unnecessary, giving the flimsy excuse that it is needed for transliterating programs written in older languages. Again, TINMAN seems of two minds, trying to decide between taking a very modern, high level approach to programming, and retaining the old ways which led us into the present state of affairs.

## 8. EXTENSION OF OPERATORS

ALGOL 68 provides a particularly elegant way of extending existing dyadic operators to new data types in the language. One can envision adding disparate structs together given the appropriate extension of the + operator. There is also the ability to define new dyadic operators, such as $$, to mean anything at all. In addition, TINMAN specifies that there must be a mechanism for associating an operator hierarchy with the new operators. ALGOL 68 includes the prio facility which permits this to be done. Unfortunately, prio can be applied to existing operators on built-in types, violating H2. It is not immediately clear how to make new data types indistinguishable from old types (E2), unless the operator priority for existing operators is fixed regardless of the type to which it is applied. If new infix operators are permitted for the new data types, precedence must be defined (in violation of H2?). In short, there are some difficulties in this area which must be resolved.

## 9. PROGRAMMABILITY

Although much harder to support by cogent and specific argument, the issue of programmability for the language envisioned by TINMAN is important to discuss. Programmability refers to the "naturalness" of the language from at least two viewpoints: learning to use the language, and remembering what it has available.

The normal pedagogical approach taken in informing even an expert programmer about a new language involves absorbing complete information about a subset of language. This technique must ultimately be applicable to all persons interested in learning

to use the language. TINMAN seems to envision a language of such intensity, flexibility, and complexity that one wonders whether there is a straightforward method for teaching the rudiments without overburdening the student with unwanted detail. The situation is reminiscent of some constructions in JOVIAL which on first encounter are arcane, to say the least.

It must be recognized that 90% of programs will use a very limited percentage of the language power. Will some of the sophisticated features of the language get in the way of average programmers? It should be clear that if a language like that envisioned by TINMAN intimidates everyone, then the language will not achieve the goals of the HOLWG effort.

## 10. ANALYZABILITY

Software analysis tools, as they are currently constructed, and as they can be projected from present technology, operate on the notion of having database representation of the (assumed syntactically correct) program in its entirety. Picture something like an "exploded parts diagram," with the entities (variables, operations, sentences/statements, blocks, etc.) as the "parts."

Once this information is available (a task accomplished by syntactic recognizers operating over the database), one can run algorithms that do many interesting and valuable things. Typically these involve functions that are only cost-effective to perform infrequently (otherwise, they would be incorporated in the compiler), or are valuable only when performed on a complete (or near-complete) software system. Acceptance testing, ferreting out hard-to-find errors, verification, control flow analyses, and quality/style investigations are some examples.

How does a TINMAN induced language simplify (or complicate) performing these kinds of functions? The answers lie in considering the rate of growth in complexity of the needed database; this is equivalent to assessing the density of semantic connectness within and among programs written in the TINMAN advocated language.

Notwithstanding some of the evidently contradictory requirements of TINMAN, the abstraction capability requirement seems to be the biggest difficulty. Consider the problems encountered with a software system which is completely expressed in user-defined abstract types. In this case (which is probably unlikely in practice and is, also, probably the worst possible situation), nothing in the program can be recognizable to the checking algorithms.

The solution to this problem, of course, is to develop algorithms that can derive the level of understanding required from the abstract definitions or which can gain access to the lower level definitions. But to achieve this may take many years after a language finds widespread use; consider that it was two decades between the introduction of the first HOLs and the development of automated program analyzers that could check coherence of programs written in them.

## 11. MISCELLANEOUS ISSUES

There are a few other points that should be cleared up. The requirement prohibiting nonrecursive routines embedded in recursive routines is reasonably trivial to achieve, but doesn't prevent some of the execution time problems associated with recursion, thereby contradicting efficiency considerations. This is still another place where the TINMAN seemed to be of two minds: the FORTRAN/COBOL mind which banishes recursion because of the required runtime support, and the ALGOL/PASCAL/LISP mind which sees the elegance of recursion despite the overhead. The present restriction (G5) is artificial.

The use of a break character in numeric literals seems to present certain implementation difficulties ("just another thing to check") and might well be dropped, particularly since numeric literals are not frequently given to more than 8 places of precision.

The issue of numeric precision for reals and integers should be thought through a bit more fully. They provide the ability to specify a representation by indicating the number of places of precision required, whether or not that precision is meaningful in the context of the computation. The experience with PL/1 has shown that serious errors can result unexpectedly from the practice of operating on numbers with different precision. This is particularly true when mixing fixed point quantities with floating point, and was the motivation concerning the earlier suggestion about eliminating fixed point artihmetic. (The problem here is the universal computer language problem--the COBOL programmers like fixed point, but it isn't needed anywhere else.)

The overhead associated with providing a variable number of arguments is considerable and goes against most language features available to programmers. One might define a certain number of built-in functions or procedures which accept a variable number of arguments, e.g., MAX, but it is not apparent how this facility might be given in general for the definition of functions/procedures by programmers.

The TINMAN requirements concerning comments indicate that the language should be both free-format and line-oriented, so that one doesn't run into the problems connected with failure to terminate a comment properly. While this can be handled satisfactorily, it presents some additional minor difficulties to the compilers which attempt to "prettyprint" the program text (as in LIS).

## 12. SUMMARY

These comments have been directed to improving the consistency of the TINMAN requirement and to suggesting

some areas in which TINMAN requirements might be tightened or loosened so that a smaller language will result. Such a language would not provide some of the machine dependent facilities of the present set of requirements, and in fact, would actively discourage such programming style since those techniques decrease program portability and understandability.

They were motivated in part by examination of CS-4, a language which is being dynamically designed to meet the TINMAN requirements. CS-4 seems unwieldy as a programming language in several respects (taken up in the subsection on CS-4 vs. TINMAN), leading one to believe that TINMAN may be asking for too much in an inconsistent way.

Given these suggestions, it should be possible to design a language (or modify an existing one) in such a way that it meets the criteria for high level languages and so that it can be used effectively by programmers.

# CHAPTER 4
# ALGOL 60

The contents of this chapter of the report involve material relating to ALGOL 60.

Section 1. Evaluation of ALGOL 60 Against Language Design Criteria

Section 2. Evaluation of ALGOL 60 Against the TINMAN

Section 3. Language Evaluation Summary for ALGOL 60

Section 4. Detailed Comparison of ALGOL 60 to the TINMAN

# SECTION 1. EVALUATION OF ALGOL 60 AGAINST LANGUAGE DESIGN CRITERIA

## 1. INTRODUCTION

This section examines the extent to which ALGOL 60 satisfies the criteria for a "good" higher-order language as presented in "Criteria for Evaluation of High-Order Languages." ALGOL 60 is seen to fulfill the requirements moderately well. However, there are significant weaknesses in most of the areas, thereby making it unsuitable for further consideration as a base language for the TINMAN.

## 2. ABSTRACTION

ALGOL 60 provides facilities for procedural abstraction through functions and procedures. Procedures and functions accept two kinds of procedures: those called by name and those called by value. Parameters passed by name are evaluated on each use of the parameter and may change dynamically. If a function or procedure name is passed by name to another or function, that function will be repeatedly called and evaluated. An expression will similarly be evaluated repeatedly; this is known as "Jensen's device." However, ALGOL 60 functions can only return values of type real, integer, and Boolean.

ALGOL 60's facilities for data abstraction are very weak, by comparison with other languages. ALGOL 60 was conceived as a language for numerical scientific programming applications and provides only the three data types of real integer, and Boolean; there is no mechanism for defining either new types or new operations on existing types. Although there are numerous extensions to ALGOL 60 which provide more sophisticated capabilities, they are not part of the Reference Report. There are no records; however, there are arrays of the three basic required types.

## 3. MODULARITY

ALGOL 60 is a block structured language in which variables may be passed between blocks. A procedure or a function with

local declarations is considered a block. As noted above, there are parameter passing mechanisms between procedures and functions. Variables which are declared local to a block are known to that block and to all enclosed blocks, with no way of limiting that access, except for renaming the variable in an inner block. A programmer must be careful in the use of variables which are global to other blocks.

In general, it is possible to achieve modularity in ALGOL 60 programs by careful use of variables and well-constrained standards for parameter passing. The availability of global variables with the ability to cause side effects in procedures and/or functions can be a technique to subvert the goals of modularity. In summary, modular programming may be accomplished in ALGOL 60, but only if the programmer takes care.

## 4. LINEAR FLOW OF CONTROL

Control structures in ALGOL 60 include the _if-then-else_, the _switch_, the _for-step-until-do_, and the _for while_, in addition to the _go to_. While these control structures, supplemented by procedure calling mechanisms (including recursion) are sufficient to eliminate the need for the _go to_, in practice they leave something to be desired. Relatively few ALGOL 60 programmers use Boolean variables with the FOR-WHILE construct to terminate loops. The tendency is to use to _go to_.

Neither the _go to_ or the _switch_ have adequate constraints on branching. Both permit transfers of control to any program location, either forward or backward in the same or a _surrounding_ block. A beter solution would constrain the switch to a group of statements as with the case statement in PASCAL, and constrain the _go to_ to _forward_ branching in the same local scope of control, i.e., the same block. Although it is possible to write programs with linear flow of control, the programmer must make an extra effort to do so.

## 5. CONTROL OF SCOPE AND BINDING

ALGOL 60 handles this TINMAN requirement quite well. Variables have an explicit scoping defined by the point of declaration. With variables declared to be of type own, there is a distinction between scope of access and scope of allocation, since such variables exist from the point of initial allocation upon initial entry to their block until program termination. However, there is no mechanism for preventing access to a variable in an inner block, except for redeclaring the same variable (possibly, with different attributes) which creates a new variable with the same name. This makes the original instance of the variable inaccessible within the block and any of its enclosed blocks.

Variables are bound to a single type at their declaration point. There are no union types and there is no way for a variable to assume a different type. Actual and formal parameters for procedures and functions must also agree in number and type.

## 6. READABILITY AND COMPREHENSIBILITY

ALGOL programs tend to be quite understandable when they are dealing with numerical scientific problems. For other applications, the programmer must be careful to select meaningful variable names (identifiers), use Boolean procedures, and isolate those routines which use the actual representation of a data object rather than its abstraction.

ALGOL 60 has been used for compilers, text editors, and many other programs which do not involve scientific computation. The limited number of control structures, the block structure, the full declaration of variables and the commenting facility all contribute to program comprehensibility. ALGOL 60 is a simple enough language that most programmers feel comfortable with it and can easily understand the intent of well-written programs. (As with all other languages, one can write poor ALGOL 60 code, but the language does not pose the intrinsic readability problems of languages such as LISP or APL.)

## 7. PROCEDURES AND DATA

ALGOL 60 makes a firm distinction between procedures and data. There is no mechanism which permits data to be executed as program text. Variables must be declared, so it is immediately apparent as to what is program text and what is data. ALGOL 60 programs which use linear control flow are thus verifiable and compilable.

## 8. LANGUAGE SIZE

By most common measures, ALGOL 60 is a relatively small language. It contains only 22 key words, less than FORTRAN or any other widely used language (BASIC excepted). The syntax is described in well under 200 productions in BNF and the complete ALGOL 60 report occupies fewer than 40 printed pages. ALGOL 60 is suitable for teaching to beginning programmers, who are able to grasp virtually all of the language concepts in a relatively short period of time.

The relatively small language size is not directly reflected in the size of ALGOL 60 compilers, however. The ALGOL 60 block structure, the two forms of parameter passing, and the availability of recursion necessitate a certain degree of run time support and a more sophisticated symbol table mechanism. Certain ALGOL 60 features, including dynamic own arrays and integer labels, have generally not been implemented because of the problems caused by these features.

## 9. ANALYZABILITY

ALGOL 60 programs lend themselves to analysis quite well, since various blocks, functions, and procedures may be instrumented without interfering with the rest of the program. One must be careful in the methods used for exiting those blocks which are being examined, so that measurements are accurate.

ALGOL 60 was also the first language in which concepts of program proving were applied. Much of the research into

programming methodology has originated with ideas which led to higher quality ALGOL 60 programs, i.e, programs that were easier to prove.

## 10. SUMMARY

The criteria used above are fundamental to language design and underlie much of the current research into programming languages and their design. It is generally accepted that the first two criteria, abstraction and modularity, are the areas in which the most important research efforts are currently being made. The shortcomings of ALGOL 60 in those respects, particularly in its limitations concerning data types, detract significantly from its acceptability.

Despite ALGOL 60's poor showing on the TINMAN criteria, any language resulting from the DoD HOL Project will undoubtedly exhibit many of the features of ALGOL 60 and its descendants. Virtually, all of the candidate langauges being studied owe a sizeable portion of their design to concepts introduced with or built upon ALGOL 60.

# SECTION 2. EVALUATION OF ALGOL 60 AGAINST THE TINMAN

## 1. OVERVIEW

The ALGOL 60 language introduced a large number of important concepts for programming languages, including such notions as definition of all variables, block structure, formal language definition, parameter passing by name and by value, and several other features. Its impact on other languages and on programming language research cannot be overstated.

Since the publication of the ALGOL 60 Report, a vast number of new languages have been developed, many incorporating a number of new concepts related to programming languages. Among these are modularity, abstract data types, restricted control structures, and record-type structures. The lack of input/output facilities for ALGOL 60 prevented the language from receiving wide acceptance and use.

The TINMAN recognizes a number of these advances as being fundamental to a "modern" programming language. Accordingly, ALGOL 60, despite its elegance, and despite the recent proposed modifications to ALGOL 60, fails to measure up to TINMAN. In fact, it falls so far short that it is not really necessary to give serious consideration to ALGOL 60 as a basis for TINMAN, since other languages have built upon the strong foundations of ALGOL while adding some of the other important notions.

The following discussion, then, is an abbreviated analysis of ALGOL 60, with emphasis on the shortcomings of the language with respect to TINMAN.

## 2. DATA AND TYPES

While ALGOL 60 is a fully typed language which supports n-dimensional arrays, it falls short on virtually all of the TINMAN specifications in this area, beyond A1. There

are no fixed point reals and no records. With respect to characters, there is a string data type, but there are no facilities for manipulating strings. There is no means for specifying the precision of fixed or floating point reals (A3). The absence of fixed point numbers rules out A4. There are no enumeration types (A5) in the sense of PASCAL scalar types. ALGOL 60 has dynamic array bounds. Both the lower and upper bounds for an array may be dynamically determined at run time upon entry to a block, violating A6. There are no records, so that there is no facility for alternative structures as with variant cases of RECORDS in PASCAL or PLEXES in LIS, so that A7 is not met.

3. OPERATIONS

Assignment and reference operations are defined for all built in data types. There are no extended data types and only a limited number of built in types (real, integer, Boolean). There are no union types (B1). It is, in general, not possible to compare two data objects (regardless of type) for identity. A Boolean cannot be compared to a real or an integer, violating B3. Reals and integers may be compared; the comparison is done in real arithmetic so that errors may result as the outcome of the conversion. Relational operations are defined for reals and integers. A different set of operations (those of propositional logic) are defined for Booleans. B4 is satisfied for integers and reals; note that there are no fixed point quantities. B5 is also met. There is no "nor" logical operation. "And" and "or" are carried out fully; any side effects are carried out--short circuits are not made. There are no scalar operations on comformable arrays. All array references must indicate the proper number of subscripts, except in the declaration of an array as a formal parameter.

Implicit type conversions are limited. Entier is a real-to-integer conversion function, but rounding is done automatically if a real quantity is assigned to an integer. The absence of exception conditions rules out B9. The absence of input/output facilities rules out B10; even the "New" ALGOL 60 Report [17] does not fully meet this requirement. Finally, there are no power sets (B11).

## 4. EXPRESSIONS AND PARAMETERS

ALGOL 60 expressions are evaluated left-to-right; thus, any side effects will be treated consistently. ALGOL 60 has 9 levels of operator hierarchy. There is occasional confusion to the reader of a program as to the levels of precedence should an expression mix logical, relational, and arithmetic operations. ALGOL 60 does not support constants (C4). The parameter rules are relatively consistent for procedures; there are no exception handling or parallel processing facilities, so that all parameters are to and from procedures. These are passed by name or by value. Some other problems in this area are discussed below.

Formal and actual parameters must agree in type. If an array is passed as a parameter, it is only necessary to declare the array and its type in the procedure heading, without declaring the number of dimensions or the size. However, the subscripts must be shown within the procedure body, so that the number of subscripts must be determinable at compile time. Thus C6 is satisfied.

ALGOL 60 has two kinds of parameters. Exception handling parameters are nonexistent. Procedure names may be passed as parameters in much the same way as other parameters. It is possible to cause some unexpected side effects by passing a constant where a value parameter is expected and then making an assignment to the formal value parameter; this is

not properly handled by all ALGOL 60 compilers. Parameter passing by name is different from procedure passing by reference. When a parameter is passed by name, it gives the ability to pass a procedure name and effect Jensen's device. The TINMAN specification wants parameters to be passed by reference and by value so as to clear up the confusion when an expression is passed by reference/name. That way, passing a procedure name must be handled by a separate class of parameters.

There are no facilities for generic procedures or for type unions. The type of all formal parameters must be specified, contradicting C8. The number of formal and actual parameters is constant and they must agree in number, violating C9.

5. VARIABLES, LITERALS, AND CONSTANTS

ALGOL 60 has no mechanism for declaring constants, nor a method for initialization of values (D1, D3). D4 is not met because of the absence of fixed point types. D5 is not met fully since there is no way to specify a range of values associated with a type, nor is there a way to define a type. D6 is not met since there is no pointer mechanism. Only D2 is met, since there is a consistent interpretation for the use of literals for Booleans and numerics.

6. DEFINITION FACILITIES

None of the definition facilities requirements are met. The user does not have the ability to define new data types (E1), except insofar as an array can be considered a new data type. This fact means that E2, E4, E5, E6, E7, and E8 are not met. E3 is satisfied, however.

7. SCOPES AND LIBRARIES

There are minor distinctions between scope of allocation and scope of access. One may declare an <u>own</u> variable,

which continues to exist from the time of first entry into the block in which it is created until the termination of the program so that the value is preserved upon subsequent entry into that block. There are some difficulties with the concept of dynamic own variables, i.e., arrays which could change their bounds. Most ALGOL 60 implementations do not support this feature and the set of modifications defined in 1976 abolishes this feature. The scope of identifiers may be determined at compile time, satisfying F3.

There are no facilities for limiting access to structures (F2).

The language provides a set of nine built-in functions, which may be viewed as comprising the beginning of a library. However, the existence of libraries is implementation dependent. Since there is no standard input/output facility, any mechanism for storing a pool of data must be machine dependent (F4, F5, F6). The lack of I/O eliminates F7.

8. CONTROL STRUCTURES

The ALGOL 60 control structures are relatively limited. G1 is partially satisfied with mechanisms for sequential, conditional, iterative, and recursive control. However, there are no facilities for parallelism, exception handling, or asynchronous interrupt handling. G2 is partially satisfied--the language has a go to statement, but labels may be anywhere in the program and there are few restrictions on branching. Furthermore, there is some problem with the use of numeric labels; some syntactic ambiguities arise when they are used as parameters to procedures. The modified ALGOL 60 Report of 1976 eliminates them. The conditional control structures are the IF-THEN-ELSE and the SWITCH. IF-THEN-ELSE is not fully bracketed and nested IF-THEN-ELSE statements must be enclosed by BEGIN-END to prevent the

ambiguity of a "dangling ELSE."

The rules concerning iterative control structures are met, since the control variable may be a normal variable whose name (but not its value) is accessible upon exhaustion of the loop. (Both the name and the value are available upon termination of the loop from within.) There is no mechanism for ending a loop other than upon completion of its scope, i.e., no embedded termination condition. Entry is restricted to the top of the loop. The step part of the FOR statement can be executed with each iteration of the loop, which can cause excessive overhead even in simple cases.

Recursive routines are permitted; there is no restriction on the declaration of internal non-recursive procedures internal to the recursive procedure. The absence of parallel processing, asynchronous interrupt handling, and exception handling eliminates G7, G7, and G8.

## 9. SYNTAX AND COMMENT CONVENTIONS

ALGOL 60 meets H1 fully. ALGOL 60 was probably the first programming language which achieved this objective; ALGOL 60 caused people to recognize the value of mnemonic variable names, uniform context-free grammars, and easily recognized forms. ALGOL 60 also meets the TINMAN requirements H2 concerning inability to modify the source language syntax, H3 concerning the ability to implement the language with a 64 character subset, H6 concerning the reservation of keywords, and H8 concerning matching of parentheses.

H4 is almost met, but there are no provisions for break characters in either identifiers or literals. Lexical units cannot, in general, be continued across lines, although some compilers treat a program as a continuous stream with no line dividers. End of line is meaningless for ALGOL 60, since there is no input/output(H5).

ALGOL 60 has a uniform comment convention, beginning with the symbol comment and ending with a semicolon ; . This is not determined by source lines, since that is not a meaningful ALGOL 60 concept.

The uniform referent criterion is not met. Procedure and function calls use parentheses, while array references use brackets.

H10 is satisfied, since the use of symbols is consistent through the language.

## 10. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

I1 is satisfied because decisions affecting program logic are explicit. The precision of reals and integers is handled by default with no mechanism for overriding them (I2). There are no compile-time facilities associated with ALGOL 60 (I3, I4).

I5 is satisfied in that the source language contains a clearly identifiable base language.

I6 is not satisfied since there is no provision for specifying language restrictions in the language definition. However, I7 is satisfied, since no language restrictions are built into the language definition. The meaning of the second part of I7 is not clear: "Language restrictions which are inherently dependent on the object environment will not be built into the ... translator."

## 11. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

ALGOL 60 does not provide any machine dependent facilities or access to machine dependent facilities within the language. There is no way to specify object representations. The efficiency of code generated by ALGOL 60 compilers varies drastically; there are some potential

overhead costs associated with the language. For example, the FOR statement is completely general in that the control variable, the step size, and the termination values may be changed within the scope of the loop. Furthermore, the step size and the termination point may be expressions involving function calls so that they must be evaluated through every iteration. Even the FOR statement

```
FOR I := 1 STEP 1 UNTIL 10 DO
```

is subject to this form of overhead in most compilers. (Some compilers provide for the case of constant step and termination and permit the programmer to specify that.)

Also, failure to use the call-by-name method of parameter passing can lead to unwanted overhead when arrays are being used between a pair of procedures. There is no way to specify whether subprograms are to be open or closed.

Thus, none of the requirements J1 through J5 are satisfied by ALGOL 60.

## 12. PROGRAM ENVIRONMENT

ALGOL 60 has been implemented in a number of environments. Although it does not require an operating system for its execution, it does require extensions for input/output to be usable in solving real programming problems. ALGOL 60 is a block structured language; there is no conceptual problem with having an inner block which is separately compiled. However, such a feature is not supported by the language.

Some ALGOL 60 environments, most notably those on Burroughs computers, provide a useful set of programming tools and aids, along with a set of extensions to the basic language which assist in monitoring variables and debugging. In general, the interface for this set of tools is an operating system so that one may save files and edit them or compile them or execute them as appropriate. Thus K3 and K4 are partially satisfied.

Much of the early work on verification and inductive assertions was based upon ALGOL 60. Assertions and other similar features may be introduced as comments. Some derivatives of ALGOL 60 incorporate assertions in a way that facilitates machine processing and checking.

13. TRANSLATORS

There are examples of ALGOL 60 translators which represent both subsets and supersets of the reference language. Burroughs' and numerous other implementations have extended the language to include input/output. Most implementations also have eliminated some of the features of the standard, including dynamic own arrays and numeric labels. Thus, both L1 and L2 are not met.

Because the translators developed for ALGOL 60 vary so greatly, it is not possible to give evaluations for ALGOL 60 on items L3-L8. It is clear that ALGOL 60 can be implemented on a number of machines. Experience has shown that the stack-oriented architecture exemplified by the Burroughs computers is particularly efficient for the implementation of ALGOL and its block structure.

While it would be possible to write an ALGOL 60 compiler in ALGOL 60, that would involve the use of integers to represent characters. Such a compiler would be particularly opaque and would be highly dependent upon the host machine.

14. LANGUAGE DEFINITION, STANDARDS, AND CONTROL

ALGOL 60 was composed of a number of features which were considered revolutionary at the time of their design. Some of them were not fully understood, a fact which resulted in some ambiguities in the original language and some trouble spots which have only been cleared up by the publication of the "New" ALGOL 60 report in 1976. However, all of the features of the language are now very well understood. There exist numerous tutorial books on the language. (C1 and C3 are met.)

ALGOL 60 was perhaps the first language to be defined unambiguously using a formal definition for the syntax. The semantics were specified in English, however. The BNF notation which is now commonly used for syntax description was first used in conjunction with the ALGOL development effort.

Maintenance of ALGOL 60 is vested in IFIP Working Group, 2.1 the ALGOL Subcommittee of IFIP TC2 on Programming. They publish a newsletter, the "ALGOL Bulletin," dealing with ALGOL 60 and ALGOL 68. They do not have the responsibility for language implementations; that lies with the individual vendors. There are no official libraries of ALGOL 60 programs. However, ALGOL 60 has long been used as the basis for the composition of algorithms intended for computer execution. Many of these algorithms have been published in collections of algorithms by ACM (originally in Communications of the ACM, now in Transactions on Mathematical Software) and in Numerische Mathematik. In general, these algorithms are not in machine-readable form and there are no standards for distribution of ALGOL 60 programs.

# SECTION 3. LANGUAGE EVALUATION SUMMARY FOR ALGOL 60

HOL EVALUATION PROJECT:
LANGUAGE EVALUATION SUMMARY FOR ALGOL 60

A. DATA AND TYPES

A01 S
A02 N
A03 N
A04 N
A05 N
A06 P
A07 N

B. OPERATIONS

B01 P
B02 N
B03 S
B04 P
B05 S
B06 P
B07 N
B08 P
B09 N
B10 N
B11 N

C. EXPRESSIONS AND PARAMETERS

C01 S
C02 P
C03 S
C04 N
C05 P
C06 S
C07 P
C08 N
C09 N

D. VARIABLES, LITERALS, AND CONSTANTS

D01 N
D02 P
D03 N
D04 N
D05 P
D06 N

E. DEFINITION FACILITIES

E01 N
E02 N
E03 S
E04 N
E05 N
E06 P
E07 S
E08 N

F. SCOPES AND LIBRARIES

F01 S
F02 N
F03 S
F04 U
F05 U
F06 U
F07 N

G. CONTROL STRUCTURES

G01 P
G02 N
G03 P
G04 P
G05 P
G06 N
G07 N
G08 N

H. SYNTAX AND COMMENT CONVENTIONS

H01 S
H02 S
H03 S
H04 P
H05 N
H06 P
H07 P
H08 S
H09 N
H10 S

I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

I01 S
I02 P
I03 N
I04 N
I05 S
I06 N
I07 S

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCE

J01 U
J02 S
J03 N
J04 N
J05 N

K. PROGRAM ENVIRONMENT

K01 S
K02 N
K03 U
K04 U
K05 N

L. TRANSLATORS

L01 N
L02 N
L03 U
L04 U
L05 U
L06 U
L07 U
L08 U
L09 P

M. LANGUAGE DEFINITION, STANDARDS AND CONTROL

M01 S
M02 P
M03 S
M04 N
M05 N
M06 P

Key: S = Satisfied
P = Partially Satisfied
N = Not Satisfied
U = Undetermined

# SECTION 4. DETAILED COMPARISON OF ALGOL 60 TO THE TINMAN

## 1. INTRODUCTION

This section outlines specific evaluations of each TINMAN requirement as they are (or are not) met by the language ALGOL 60.

Page number references when given refer to the basic reference document for the language: P. Naur (Editor), "The Revised Report on the Algorithmic Language ALGOL 60," Communications ACM, January 1963.

A. DATA AND TYPES

A1. Typed Language
Satisfied

A2. Built-in Data Types
Not Satisfied

No records.

A3. Precision Declaration
Not Satisfied

No precision specification.

A4. Fixed Point
Not Satisfied

Integers only; no fixed point decimals

A5. Enumeration Sense of Character Sets
Not Satisfied

A6. Arrays
Partially Satisfied

An array of variable size may be declared in a procedure or function, then allocated at execution time depending upon the value of dynamically determined variables. Furthermore, such arrays can be deallocated and reallocated with a different size during program execution. However, the number of dimensions and typing of arrays is fixed.

A7. Record Equivalence
Not Satisfied

No records.

B. OPERATIONS

B1. Assignment
Partially Satisfied

B2. Equivalence Operator
Not Satisfied

ALGOL 60 requires = for numerics and ≡ for Booleans. Comparison between numerics and Booleans is an error.

B3. Relationals Defined
Satisfied

Note: there is no enumeration.

B4. Arithmetic Operations
Partially Satisfied

No fixed point. / accepts operands of real or integer; ÷ applies only to two integers.

B5. Truncation/Rounding
Satisfied

B6. Boolean Operations
Partially Satisfied

Provides ∧, ∨, >, no nor; does not short circuit.

B7. Scalar Operations
Not Satisfied

Must index elements

B8. Type Conversion
Partially Satisfied

No fixed point

B9. Range Exceptions
Not Satisfied

No exception handling for run-time truncation; however, run-time truncation should not occur. No numeric ranges.

B10. I/O Operations
Not Satisfied

No I/O facilities in ALGOL 60 - I/O defined in New ALGOL 60 Report.

B11. Power Set Operations
Not Satisfied

No power sets of enumeration provided.

C. EXPRESSIONS AND PARAMETERS

C1. Side Effects
Satisfied

C2. Operand Hierarchy
Partially Satisfied

Nine levels of operator hierarchy

C3. Expressions Permitted
Satisfied

C4. Constant Expressions
Not Satisfied

No specification for constants

C5. Parameter Rules
Partially Satisfied

The declaration of an array within a procedure need not show dimensions; no exception handling, no type declarations; no parallel processing.

C6. Type Agreement
Satisfied

C7. Formal Parameter Kinds
Partially Satisfied

Call be value and call by name. Procedures as parameters are handled as call by name. No class for specifying control action when exceptions occur.

C8. Formal Parameter Specifications Optional
Not Satisfied

Type must be specified. Dimension and number of dimensions need not appear in formal parameter list. However, use of array within procedure yields number of dimensions.

C9. Variable Number of Parameters
Not Satisfied

One can pass arrays of variable size, but no argument lists of variable length.

D. VARIABLES, LITERALS AND CONSTANTS

D1. Constant Value Identifiers
Not Satisfied

D2. Numeric Literals
Partially Satisfied

No specifications for precision of real literals between object machines.

D3. Initialization
Not Satisfied

D4. Ranges and Stepsize
Not Satisfied

D5. Variable Types
Partially Satisfied

No records, but true within data types supported by language.

D6. Pointer Variables
Not Satisfied

No pointers

E. DEFINITION FACILITIES

E1. Type, Operator Definitions
Not Satisfied

E2. Consistent Use
Not Satisfied

No defined types

E3. No Default Declarations
Satisfied

E4. Extend Existing Operators
Not Satisfied

No new types

E5. Type Definitions
Not Satisfied

E6. Data Definitions
Partially Satisfied

Arrays only; no records, power sets, enumeration, discriminated union.

E7. No Free Union
Satisfied

E8. Type Initialization and Allocation
Not Satisfied

Not applicable

F. SCOPES AND LIBRARIES

F1. Allocation and Access
Satisfied

F2. Limit Access Scope
Not Satisfied

No type definitions

F3. Static Scope Determination
Satisfied

F4. Libraries Available
Undetermined

F5. Library Contents
Undetermined

Implementation dependent--not prohibited by Reference Language.

F6. Libraries and Compools
Undetermined

F7. Machine Dependent Interface
Not Satisfied

No I/O

G. CONTROL STRUCTURES

G1. Control Structures
Partially Satisfied

No parallelism, asynchronous interrupts, or exception-handling.

G2. GOTO
Not Satisfied

Can branch out of local scope.

G3. Conditional Control Structures
Partially Satisfied

Cannot branch on discriminated union; else not required with if...then.

G4. Iteration Control
Partially Satisfied

Termination at loop beginning or via branch out of loop; control variable need not be local.

G5. Recursive Routines
Partially Satisfied

Can define procedures within recursive procedure.

G6. Parallel Processing
Not Satisfied

G7. Exception Handling
Not Satisfied

No exception-handling

G8. Synchronization Primitives
Not Satisfied

Not specified in language; OS dependent

H. SYNTAX AND COMMENT CONVENTIONS

H1. Syntax Characteristics
Satisfied

H2. No Syntax Extensions
Satisfied

H3. Source Character Set
Satisfied

H4. Literals and Identifiers
Partially Satisfied

No break character for use internal to identifiers or literals.

H5. Continued Lexicals Units
Not Satisfied

No End-of-line in literal strings

H6. Keywords
Partially Satisfied

Implementation dependent - about 22 keywords + those added for I/O in some implementations.

H7. Comment Convention
Partially Satisfied

Requires explicit comment terminator.

H8. Unmatched Parentheses
Satisfied

H9. Uniform Referent
Not Satisfied

[ ] for subscripts; ( ) for function calls

H10. Consistency of Meaning
Satisfied

I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

I1. Defaults in Program Logic
Satisfied

I2. Default Override
Partially Satisfied

Programmer cannot override defaults.

I3. Compile-time Parameters
Not Satisfied

No compile-time facilities

I4. Object Environment Compile-time Parameter
Not Satisfied

I5. Language Kernel
Satisfied

I6. Translator Restrictions Defined
Not Satisfied

No translator limits defined

I7. Object Environment Restrictions
Satisfied

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCE

J1. Efficient Code
Undetermined

Dynamic own arrays are treated differently.

J2. Optimization Invisible
Satisfied

J3. Access to Machine Language
Not Satisfied

ALGOL 60 does not permit machine code insertions - not a systems programming language.

J4. Implementation Representation Tradeoffs
Not Satisfied

J5. Open/Closed Implementations
Not Satisfied

# CHAPTER 5
# ALGOL 68

The contents of this chapter of the report involve material relating to ALGOL 68.

Section 1. Evaluation of ALGOL 68 Against Language Design Criteria

Section 2. Evaluation of ALGOL 68 Against the TINMAN

Section 3. Language Evaluation Summary for ALGOL 68

Section 4. Detailed Comparison of ALGOL 68 to the TINMAN

# SECTION 1. EVALUATION OF ALGOL 68 AGAINST LANGUAGE DESIGN CRITERIA

## 1. INTRODUCTION

This section examines the extent to which ALGOL 68 satisfies the criteria for a "good" higher order language as presented in "Criteria for Evaluation of High-Order Languages." ALGOL 68 is seen to fulfill the requirements quite well. The primary areas of weakness of ALGOL-68 are the absence of abstract data types and the size of the language.

## 2. ABSTRACTION

ALGOL 68 provides excellent facilities for procedural abstraction. It is possible to declare a proc which accepts two types of parameter: those called by value and those called by ref. The value returned by a proc may be an object of any mode, whether built-in or user-defined.

The facilities for operator and operand abstraction are also quite good, although ALGOL 68 does not support the concept of encapsulating the set of operations on a data type with the representation of the type (abstract data type). First, it is possible to define new operators as infix symbols in much the same way as one defines procedures, and to associate a priority with these operators. Thus, one could extend ALGOL 68 for sets, for example, and include a set of dyadic operators (or extend existing operators) to carry out the basic set of operations of union, intersection, complementation, subset testing, etc.

One can define new data types in a limited way. Record classes are provided (struct) with the generality that structures may be declared within structures may be declared within structures. Existing operators are extended to apply to substructures as appropriate and there are standard initialization facilities for objects of mode struct.

It would be relatively straightforward to extend ALGOL 68 to provide an abstract data type facility comparable to that of

Euclid (for example). What would be required would be an exports clause to identify the visible names, a rep clause which declared the representation of the object, and an ops clause to encapsulate the associated operators, thereby providing the extended mode in an abstract format.

3. MODULARITY

ALGOL 68 is a block structured language in which variables may be passed as parameters (reference or value) or may be shared through global declarations. There is no obvious mechanism whereby items declared in one block may be kept private from those blocks nested within it, as with the PRIVATE declaration of LIS.

While it is possible to achieve modularity in ALGOL 68 programs through the use of parameter passing, there are also a number of mechanisms for subverting modularity, caused primarily by the freedom to access global variables and by the capability to produce side effects in a value-returning proc. For example, the proc call f(x) may return a value for f, but could also change the value of x. In general, though a proc may represent a procedural module.

As with other block structured languages (ALGOL 60, PL/I) and quasi-block structured languages (PASCAL), modularity can be achieved, but only with some care on the part of the programmers. As with the other languages, there is no method for establishing "data modules."

4. LINEAR FLOW OF CONTROL

The control structures of ALGOL 68 include the go to statement, with no serious restrictions on its use. A go to can designate any label within the present scope of a program or within any enclosing scope. It is expected that the need for the goto will be quite limited, however, since ALGOL 68 supports a number of other control structures, including an if-then-else, various forms of a do statement, a case statement, and a while statement.

In general, the control structures for ALGOL 68 will not require the use of the go to with any frequency.

## 5. CONTROL OF SCOPE AND BINDING

Scope and binding of variables is properly managed in ALGOL 68, with variables declared upon entry to a block. Space is allocated upon block entry and deallocated upon exit, unless the object is declared with a heap generator, in which case it remains in existence until the program is terminated.

Variables are bound to a mode upon their creation and cannot change that mode. It should be noted that the mode may be a union of basic modes in the language, e.g., mode rent = union (real, int).

The scope of each declaration is unambiguous and visible from the program's static listing.

## 6. READABILITY AND COMPREHENSIBILITY

ALGOL 68 programs are generally comprehensible. Two language features which tend to detract from this readability should be cited, however. First, the symbols begin and end may be replaced by matched parentheses, and the construct if A then B else C may be replaced by (A|B|C). Second, some operations, such as plusab (equivalently :=+), conceal the fact that an assignment is made to a variable.

On the other hand, ALGOL 68 is exceptionally consistent as a language; an ALGOL 68 programmer quickly becomes familiar with major constructs and their use. Accordingly, it becomes relatively easy to understand ALGOL 68 programs. The most significant difficulty is caused by the size of the language and by the ability to define new operators. If a programmer has made liberal use of the permissible abbreviations for keywords and has defined a significant number of dyadic operands and/or changed the priority of existing operands, the structure of an ALGOL 68 program can become rather difficult to comprehend.

It seems that ALGOL 68 programs are somewhat less readable than programs in PASCAL, but rather more comprehensible than a comparable program in PL/I or FORTRAN.

## 7. PROCEDURES AND DATA

ALGOL 68 maintains a distinction between procedures and data. It is not possible to modify ALGOL 68 instructions or to read data and then execute it, as is the case in COBOL, MUMPS, SNOBOL4, and several other languages. Since variables must be declared, ALGOL 68 always makes it clear as to what is program text and what is data.

## 8. LANGUAGE SIZE

ALGOL 68 is a moderately large language in terms of the number of syntactic units required to represent the grammar. However, the number of keywords in the language is relatively small, well under 100. ALGOL 68 is defined with a W-grammar, which includes both syntactic and semantic elements. The beginner would have significantly more difficulty comprehending the ALGOL 68 report than other reports written in BNF or a less formal notation.

The size of ALGOL 68 is somewhat deceiving, because only a relatively small number of individual concepts are included. An excellent and largely complete tutorial booklet on the language is shorter than 200 pages, even shorter than the more formal reference report.

More substantial than the language size is ALGOL 68's own terminology, which will be particularly foreboding to a person accustomed to programming in other languages. After one gets past the initial barriers to comprehension of the language, the language is surprisingly manageable intellectually. The language is exceptionally smooth, with only a few inconsistencies or special requirements, e.g., casting. However, ALGOL 68's size presents substantial problems for the implementor, who cannot easily use the syntactic processing techniques that have been used for languages defined in BNF.

## 9. ANALYZABILITY

ALGOL 68 programs can rate fairly high on analyzability if they are written using the powerful control constructs above. The language contains the concept of "pragmats," denoted by pr, which are treated as comments. However, pragmats may contain assertions and could be used to generate verification conditions for program provers.

Similarly, test data can be generated and execution can be monitored in a relatively straightforward way in ALGOL 68. Thus, ALGOL 68 imposes few barriers to automated analysis.

## 10. SUMMARY

Although ALGOL 68 has some shortcomings when measured against this set of criteria, it also possesses the expressiveness and conceptual integrity which are at the heart of programming language design. Because the language features are largely orthogonal, it would be a straightforward exercise to delete certain features and propose others without drastically altering other language features. Thus, one immediately sees how to add abstract data types, sets, fixed point numbers, or other data types, while reducing the generality of the go to and eliminating such things a flex arrays and certain operators which make the language less satisfactory. As a result, it is extremely tempting to begin with the notions of ALGOL 68 as the basis for meeting the criteria of TINMAN, rather than making minor changes in a language which was constructed in an ad hoc manner to conform to TINMAN.

# SECTION 2. EVALUATION OF ALGOL 68 AGAINST THE TINMAN

## 1. OVERVIEW

ALGOL 68 provides a number of the facilities which are important in the creation of a programming language supporting a systematic approach to software development. With respect to the criteria of the TINMAN, ALGOL 68's greatest strengths lie in full declarations of data objects, the block structure of the language, the ability to extend binary operators to newly defined data types, support for virtually all of the basic data types, and a sharply defined concept of data type (termed mode). Its major weaknesses are the lack of abstract data types, the failure to support the uniform referent concept, and the nondeterministic evaluation order of expressions in ALGOL 68 collateral clauses. Going beyond the criteria of TINMAN, one can also criticize ALGOL 68 for some of the features which present implementation difficulties and conceptual difficulties for someone learning the language. In particular, the ALGOL 68 terminology and the presentation of the reference document are quite formidable, even for those with substantial exposure to programming languages.

Notwithstanding these points, ALGOL 68 must be considered a strong candidate (as the basis) for a DoD standard, with appropriate extensions for abstract data types and restriction on the use of collateral clauses. This acceptance would clearly have to be supported by extensive tutorial documentation and training. The following sections give an overview of the compatibility between ALGOL 68 and TINMAN, followed by a detailed comparison between the two. Emphasis is given to those aspects of the language which are particularly well handled by ALGOL 68 and to those which are serious detriments to the language with respect to TINMAN.

## 2. DATA AND TYPES

ALGOL 68 is a strongly typed language in which there are few unexpected conversions from one type to another. The six types of conversions are termed dereferencing, widening, rowing, deproceduring, voiding, and uniting. Dereferencing is done when an assignment statement of the form:

```
a := b
```

is performed, where a and b are declared to be of the same mode, real for example. b has mode ref real, as does a. In order to assign the value of b to a, it is necessary to obtain the value of b, thereby dereferencing b and producing a value of mode real.

Widening would occur if we had declared

```
real a, int b
```

with a := b, b would be of mode ref int. The integer value would have to be coerced to a real value to complete the assignment; this is termed widening. The reverse operation, had we declared

```
int a, real b
```

and written a := b, would be illegal, since dereferencing and widening would not apply. It would be necessary to use a monadic operator such as round or entier to make a proper assignment, e.g.

```
a := round b
```

Uniting permits an object to belong to a type union, e.g.

```
mode m = union (int, real)
```

Had we declared

```
m a,b
```

using this new mode, then the assignment a := b would work

if b were either int or real; a would then assume the corresponding mode.

All of these coercions among types are compatible with Tinman requirement A1, and also with B8 (except for the nonexistent fixed point types).

ALGOL 68 supports built-in modes of int, real, char, string, Boolean, bits, and bytes along with arrays of these modes and structs which provide a record-handling facility. The int and real specification may be preceded by one or more instances of the words short or long to impart information about the size of the object or the required precision. The interpretation of these words is strictly up to the implementor, though (and could be ignored). The ability to specify precision is not comparable to that facility as it exists in PL/1. (Precision specifications in PL/1 are a frequent cause of programming errors and unexpected results as a result of truncations of leading digits and roundoff errors.)

ALGOL 68 does not explicitly support fixed point arithmetic (as does COBOL, for example). However, one could devise a set of procedures and extend the available (and meaningful) binary operators as part of the standard prelude to add this facility. (Fixed point arithmetic is also a common source of errors in numerical computation.)

While ALGOL 68 requires declarations of arrays, with an explicit number of dimensions and bounds, it is also possible to declare arrays with the property flex which permits the bounds of those arrays to be changed dynamically. Flex arrays clearly violate the intent of TINMAN; they could be deleted from the language without major impact on other language features.

3. OPERATIONS

The ALGOL 68 language compares quite favorably with the TINMAN specifications regarding operations. Assignment

works in the desired manner (B1), relational operators are properly defined (B3), the basic built-in arithmetic operations are present for real and int values (B4), truncation and rounding will operate properly (although the implementation of the language may cause roundoff errors as a result of performing decimal arithmetic with a binary floating-point device (B5)), and all of the desired Boolean operations other than "nor" are present (B6). ALGOL 68, like PL/1, permits assignments of matching array rows or records without explicit indexing. Thus, the declaration

```
[-10:10] int a,b;
```

permits us to write

```
a := b.
```

Short circuit mode is not defined in the language, although at least one implementation of ALGOL 68 gives the user the ability to follow that alternative. The language design decision was to permit side effects associated with the evaluation of a Boolean expression, i.e., values changed as the result of assignments, function calls, or procedures.

ALGOL 68 has an extremely powerful facility for defining new binary operators and for extending binary operators to new types. While the concept of equality is defined for the built-in types, it is not automatically defined for new modes. However, if we introduce a new mode foo, we may then write

```
op = = (foo a,b)  bool;
```

and proceed to define the concept of equality for objects of mode foo, thereby making it possible to compare objects of type foo for identity. This notion can be extended to compare any two data objects regardless of type (B2). The number of definitions that would have to be made can become combinatorially large, since the operator must be explicitly defined for every conceivable pair of types. Thus for all practical

purposes, ALGOL 68 does not meet the Tinman requirement. A more realistic requirement would be that the equality test exists between any two objects of the same mode; this is in conformance with the equality test present in the language CLU and is quite workable. The intelligent programmer should not be in the position of wanting to compare objects of different mode as a general rule: int/real and char/string are typically the only meaningful mixed mode comparisons.

ALGOL 68 provides operations beyond those specified by Tinman. It is often the case that one will write

```
I := I + 1
```

in a program, where I is of mode int. ALGOL 68 provides the abbreviated operator +:= (or plusab) permitting this to be written as I plusab 1. Similar operators exist for all of the fundamental arithmetic operations so that one could write

```
p timesab q + 5
```

which means

```
p := p *(q+5).
```

The assignment operation is hidden by the operator so that the reader of the program does not immediately observe the assignment. The specification of these operators requires that the left-hand-operator be of type ref int or ref real so that

```
5 + q plusab p
```

is illegal since 5+q does not coerce to mode ref real or ref int. The operators plusab, minusab, timesab, divab, overab, and modab should be removed from the language in order to conform to the specifications of Tinman.

ALGOL 68 frequently permits more than one symbol to stand for an operation, e.g. +:= and plusab or (worse) mod, ÷*, ÷x, %*, and %x. While this is sometimes necessitated

by constraints on the size of the character set, a more workable approach would define the language in terms of the ASCII character set and provide a single symbol for each. The interchangeability of matched parentheses with begin and end, although justified by the places in which a closed clause may appear, can be particularly annoying. Programming standards would have to be established in order to produce readable programs and identifiable blocks in ALGOL 68 programs.

4. EXPRESSIONS AND PARAMETERS

ALGOL 68 has a serious conflict with Tinman with respect to C1. The order of evaluation for arguments of an expression is not specified in the ALGOL 68 Report. In fact, it is deliberately left unspecified, in order to give freedom to the implementor to make certain optimizations. If one wrote

```
S := F(X) + G(X)
```

there would be no guarantee of the order of evaluation of F and G. Should either of the functions change the value of X (a side effect), the value of the function would be different (potentially) for the two orderings of evaluation. Collateral elaboration is a fundamental concept in ALGOL 68, and one must go to considerable care in programming to avoid the effects of collateral elaboration rather than serial elaboration. A fairly strict programming discipline regarding function calls and the implications of side effects would be required, since imposition of a left-to-right ordering would fundamentally change the language, necessitating significant changes in the language definition.

ALGOL 68 has more levels of operator hierarchy than are generally considered to be desirable. (In contrast with LIS, which has 4, ALGOL 68 has 10 levels, with the plusab

(etc.) operators at level 1 and some of the monadic operators like upb and lwb at level 10.) Significant also is the fact that the ALGOL 68 programmer can change the priority of any operator through the prio declaration, in addition to specifying the priority of newly defined binary operators. Thus, the standard prelude of the language could be modified to define a smaller number of levels of operator hierarchy. However, the large number of levels of hierarchy are a result of the concept of collateral elaboration. If all operators were at the same level of hierarchy, the expression

```
a + b x c ** d
```

could very well evaluate differently under different implementations of the language. (This is a tradeoff factor in language design.)

ALGOL 68 meets the other requirements with respect to expressions and parameters quite well. Expressions are permitted wherever both constants and references to variables of that type are allowed (C3), although there are instances of potential ambiguity which must be resolved by casting. Consider

```
begin int i := 0, k := 1;
ref int ptr := i;
     ptr:= k;
     print (i);
end;
```

The assignment ptr := k can cause trouble, since ptr is of mode ref ref int and k is of mode ref int. The standard dereferencing must not occur here, since ptr points to i, and the assignment would cause i to be assigned the value 1, which is quite different from the evident intent. Thus, the assignment statement must be replaced by

```
ref int (ptr) := k;
```

casting the type of the result.

Rules for parameters are uniform: ALGOL 68 supports calls by reference and by value in a manner corresponding closely to PASCAL. (ALGOL 68's call by reference is different from the ALGOL 60 call-by-name concept.) Operands to which assignments can be made carry with them the symbol ref to designate call-by-reference. Hence

```
proc add = (int a, int b, ref int c) void;
     begin c := a + b end
```

is legal. Note that if c was of mode int rather than mode ref int, the assignment statement would not be permitted. This is a problem in FORTRAN which does not make the proper differentiation; one could write

```
SUBROUTINE ADD (A,B,C)
C = A+B
RETURN
END
```

and legally write CALL ADD (2,3,4), which is not permitted in ALGOL 68, in which the type of the actual parameter is compared with that of the formal parameter. In this instance, 4 would not be coerceable to mode ref int and an error would be signalled.

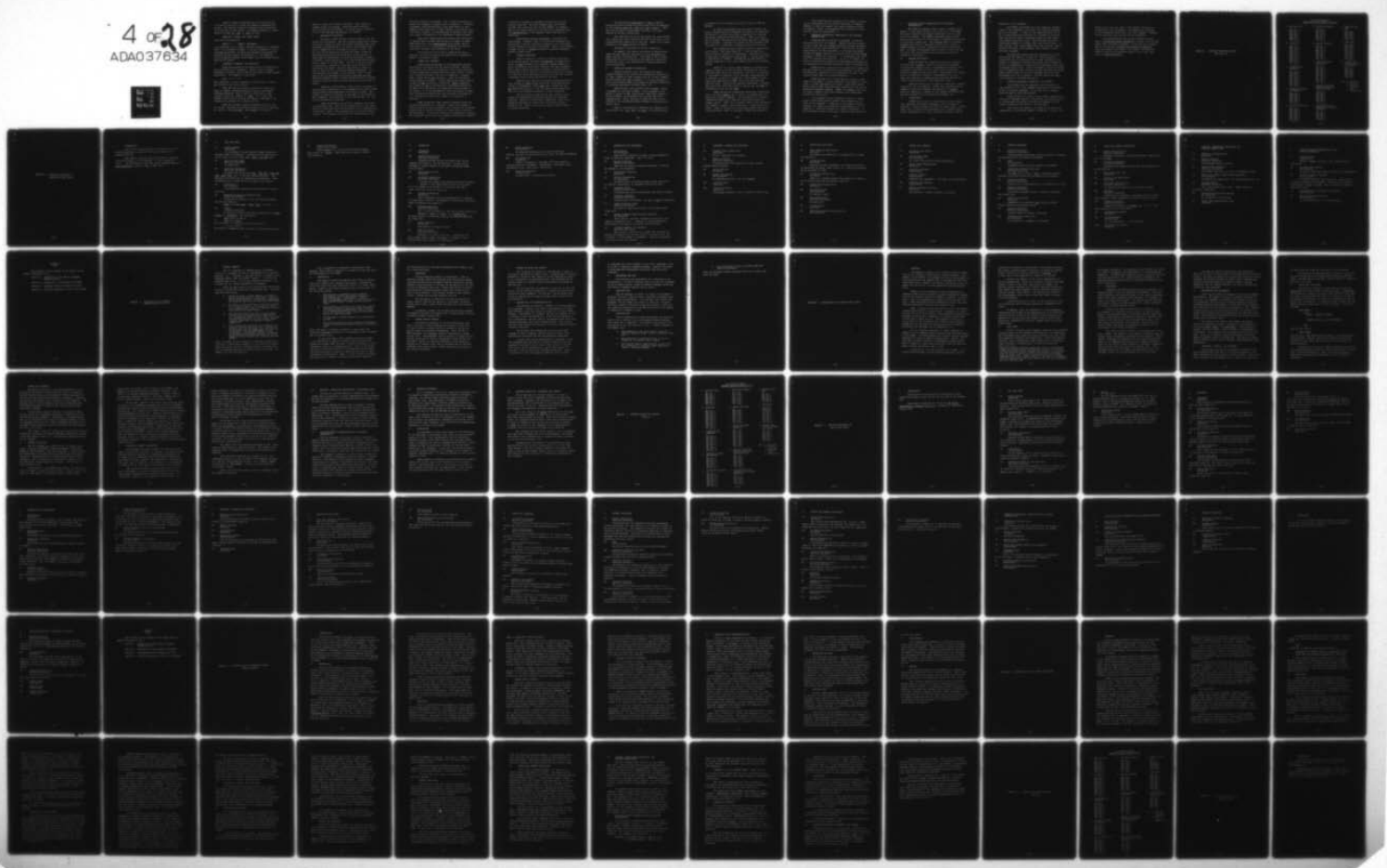
ALGOL 68 permits procedure names to be passed in the same manner as other parameters and even returned as results. They do not constitute a separate class of parameters in the sense of TINMAN's C7. Also, ALGOL 68 has no provision for exception-handling or for passing exception conditions as parameters. Extension of the language for exception-handling is quite complicated in order to control variables and their scoping properly.

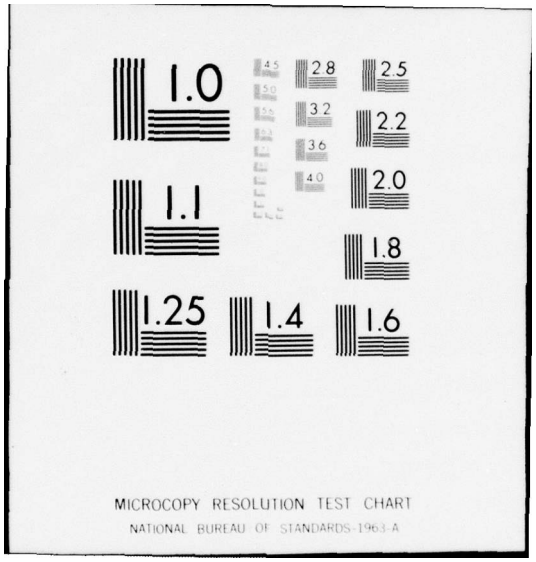
Variable length parameter lists cannot be passed in ALGOL 68. However a pointer can point to a linked list of objects so that one can pass the pointer as a parameter, giving the effect of passing a variable number of items when only passing one.

AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

4 OF 28
ADA037634




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Types of formal parameters must also be specified (in conflict with C8), as must the number of dimensions for arrays. The concept of generic procedures may be achieved by making use of type unions. A procedure designed to handle objects of mode int, real, or bool can have

```
mode mess = union (int, real, bool)
```

and also the declaration

```
proc p = ( ... mess m, ref mess q, ... ) ...
```

in a parameter list to permit treating parameters of differing type. If a new mode is added, it is only necessary to alter the mode declaration, and make any necessary discrimination based on the mode within the procedure p. The ALGOL 68 solution appears somewhat more elegant than the TINMAN requirement C8 will permit.

## 5. VARIABLES, LITERALS, AND CONSTANTS

ALGOL 68 provides for the definition of constants and the initialization of variables. There is only a slight difference in syntax, which could be the cause of some programming errors. For example, we can write

```
int i = 2, j := 3;
```

which declares i to be a constant of value 2 and j to be an int variable with initial value 3.

ALGOL 68 does not provide a facility to limit the range of numeric variables along the lines of the PASCAL subrange mechanism. Such a facility would lead to some confusion about the concept of a mode. It is clear the TINMAN's specification E7 does not require a subrange to be treated as a new mode (as is done in CS-4).

ALGOL 68 provides a pointer mechanism which is safe in its use. One does not declare an object to be of mode ref, but rather of mode ref mode, where mode is the mode of

object to which the variable can point. This cannot be changed within the program. ALGOL 68 permits multiple level pointers as does PASCAL. A variable may be declared to be of mode ref ref ref ref int, for example.

6. DEFINITION FACILITIES

The user of ALGOL 68 has a limited facility for the definition of new modes. One may introduce modes which include structures, unions of types, structures dependent upon other variables (corresponding to the variant case RECORD structure of PASCAL but not as general as plexes in LIS). One may also define operations on those new modes, either by defining procedures or functions which accept objects of those modes as parameters, by defining new dyadic (binary) or monadic (unary) operators, or by extending existing operators to apply to the new modes. The set of built-in modes is sufficiently extensive to provide a good facility for introducing new modes. The only mode which is absent is that of sets with the associated powerset mechanism. If a set mechanism was present in the language, there is some chance that the designers would have eliminated pointers from the language, since it would then be possible to define recursive data structures without pointers (as described by Hoare).

ALGOL 68 is particularly attractive in the manner in which existing operations in the language may be used to make defined modes indistinguishable from built-in modes. The concept of operator extension to new modes is not present in many languages. New modes may be initialized (E8), but cannot be automatically finalized.

ALGOL 68 supports the notion of abstract data types as specified by E5. There is no encapsulation of the operators on a given mode with the definition of the representation of the mode. Hence, the representation of the mode remains visible. The necessary facilities for mode definition and

operator extension are present, and it would be feasible to extend the language to support such a capability. A careful programmer could achieve much of the same effect as with abstract data types even in their absence; one does not have the option, however, of incorporating an abstract type (such as a stack) in hardware or as a separately compiled module.

Type union capabilities are present with the ability to declare that an object may be one of a number of modes. Mode definition is by discriminated union rather than by free union; i.e., it is possible to determine the mode of the object through a case statement at any point.

The definition facilities of ALGOL 68 are quite powerful, but fail to provide for power sets of enumeration modes or for abstract types.

## 7. SCOPES AND LIBRARIES

ALGOL 68 has mechanisms for distinguishing between the scope of allocation and the scope of access. Scope of access rules are similar to ALGOL 60, with a block structure in which names may be redefined within inner blocks. Control over allocation differs from that of ALGOL 60. A trouble spot in ALGOL 60 was own variables: their initialization versus dynamic allocation. These concepts have been replaced by loc and heap allocation in ALGOL 68. Heap encompasses the concept of own by providing for continuing existence of an object with its value, but preserving limited access to the object. It is not possible to limit the access to separately defined structures other than through this mechanism; i.e., no abstract types limit access to a fixed set of operators.

ALGOL 68 does not make lengthy statements about the run-time environment for a program. There are a number of built-in functions and procedures, and it should be a straightforward task to provide libraries of other useful functions and procedures. As with most high level programming langauges, the translator would exist on a computer system which supports

a library of programs on secondary storage which can be directly executed. The base language does not provide straight text insertion (F5) as does RATFOR, for example. Many of the issues implicit in F5 and F6 are dependent upon the implementation of the language rather than upon its definition.

ALGOL 68 has machine-independent procedures for input/output. However, there is no built-in mechanism for associating a logical file with a physical file, which must be done through a job control mechanism. There is no inherent language feature to prevent interface to machine dependent hardware capabilities.

8. CONTROL STRUCTURES

ALGOL 68 supports a variety of control structures, including the go to, bracketed if-then-else and case statements, a while statement and a general looping capability which incorporates controlled iteration (the for statement). When a for is used, the control variable is not accessible outside the loop it controls; the control variable is treated as a strictly local declaration within the loop, and is given a constant value on each iteration of the loop.

ALGOL 68 also supports concurrency (pseudo-parallelism) through the down and up facilities, which provide the concept of semaphores, called sema mode in ALGOL 68. The par declaration prior to a set of closed clauses specifies they are to be treated as parallel in their execution.

The facilities of ALGOL 68 are very primitive with respect to asynchronous interrupt handling and exception handling. Except for input/output conditions, there is no exception handling for faults such as division by zero, overflow, etc. There is no way for users to easily define or signal exceptions.

The bracketed if-then-else and case statements provide full partitioning for choices and avoid the ambiguity of the dangling else (if A then if B then C else D). They may also be used to discriminate the type of an object defined as having a united mode.

The conditions of G4 are met except for loop termination, which must occur either at the end of the loop or via an unconditional branch out of the loop. There is no correspondence to the leave feature of BLISS.

Recursion is provided in its full generality.

In summary, the control structures of ALGOL 68 are among the most powerful available in any programming language, possibly excepting PL/I. The flaws in PL/I control structures lie in the looping (which ALGOL 68 treats more effectively) and in the unconditional branching permitted with ON-conditions.

9. SYNTAX AND COMMENT CONVENTIONS

ALGOL 68 permits a well structured and highly readable program to be created with appropriate indentation of lines (though not done automatically as in LIS), mnemonic variable names, and ample comments. It is also possible to write less readable ALGOL 68 programs by ignoring indentation, by choosing meaningless variable names, etc.

ALGOL 68 has a well-defined formal grammar, called a W-grammar or two-level grammar. Such a grammar is not parsed as easily as BNF, and presents some minor implementation problems; the W-grammar does provide additional language power. Early versions of ALGOL 68 were context-sensitive, but the Revised ALGOL 68 Report is syntactically context free.

Some of the keywords in ALGOL 68 are themselves abbreviations (e.g., int, bool, and sema), but abbreviations

of keywords are not possible in the PL/I sense of DCL and DECLARE.

An ALGOL 68 programmer has the power to modify the source language syntax by changing the priority of operators. The ability to define priorities is essential for creating new dyadic operators for defined modes. There is no easy way to prevent this operation from being applied to existing built-in modes. Limiting this capability thus detracts from the usability of ALGOL 68's language extension facilities.

The language defines formation rules for identifiers and literals. The blank character " " may be used as a break character internal to identifiers. Character literal strings may be separated by quote marks ", so that "abc""def" is equivalent to "abcdef". Spacing may be used freely to improve readability; only basic symbols cannot have internal spaces.

ALGOL 68 is a fairly large language in terms of the complete number of constructs which are permitted, and the different modes which are supported. This fact is reflected in the number of keywords in the language (approximately 50 in contrast to only 22 in ALGOL 60). This number is quite small when compared with COBOL, which has more than 300 reserved words. Keywords are generally meaningful, although many objections have been raised to use fi, od, and esac as terminators of if, do, and case.

ALGOL 68 has four commenting conventions, permitting use of the word comment, co, #, and ¢, in matched pairs to begin and terminate comments. The availability of multiple terminators makes use of specific characters within the comment easier. There is no automatic end-of-line terminator for comments, however, since ALGOL 68 is a free-format language. Input is treated as a continuous stream; the end-of-line requirement seems to contradict TINMAN requirement H1.

While ALGOL 68 has uniformity with respect to matching of parentheses and well-defined meanings of language symbols, it does not support the uniform referent concept. For example, arrays are designated with brackets [], while function calls use parentheses ().

## 10. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

ALGOL 68 has no defaults in programs, unlike the situation with PL/I for example. Objects must be declared to have a mode and the language processor assigns storage on the basis of that mode declaration. The only way in which these defaults can be overridden is to declare new modes through the struct capability using the built-in modes bytes and bits to define the structure. It is generally not practical to override these defaults. In fact, to support the notion of abstraction properly the programmer should not have the opportunity to override the default object representation.

ALGOL 68 has no facilities for conditional compilation. The macro facility in ALGOL 68 is non-existent, as one cannot even declare a subroutine to be inserted as an open subroutine rather than as a closed subroutine.

ALGOL 68 also fails to set explicit limits on such items as the number of array dimensions, the length of identifiers, or other implementation issues, leaving each implementor free to make those decisions, possibly at the loss of portability between programs. MUMPS, for example, has a set of limits (termed a Level One Standard) that establishes portability conventions.

It is arguable whether ALGOL 68 contains a "simple clearly identifiable base or kernel which houses all the power of the language." ALGOL 68 is an extremely powerful language, but there is considerable disagreement as to its simplicity. ALGOL 68 features are largely orthogonal, i.e. each language feature contributes a single unique capability.

## 11. EFFICIENT OBJECT REPRESENTATION AND MACHINE DEPENDENCIES

The ALGOL 68 language report makes no comments with respect to the issues raised in this section of the TINMAN. The source language requirements indicated by J3, J4, and J5 are not present. The TINMAN requirement about optimizations and the effect of the program is handled by the notion of collateral elaboration. The language makes no statement regarding the order of expression evaluation to assure a program remains "correct" regardless of any optimizations made. An ALGOL 68 programmer must be aware of this fact and not write code which depends upon a particular order of expression evaluations.

## 12. PROGRAM ENVIRONMENT

ALGOL 68 makes few assumptions about the implementation of the language. While it is certainly desirable to incorporate ALGOL 68 into a programming environment with a powerful complement of tools, this issue is separate from the language definition issues. Furthermore, the availability of subroutine libraries, programming tools, etc., imply the existence of sophisticated operating system facilities, in contradiction to the assumption of K1.

ALGOL 68 permits the inclusion of assertions, compiler directives, etc. as comments. Rather than treating them as comments, ALGOL 68 introduces the concept of a pragmat, which is treated as a comment by the compiler but which might provide information for verification programs, etc.

## 13. TRANSLATORS

TINMAN states that all translators for the language must implement the complete language, with no subsets or supersets. ALGOL 68 makes provision for these cases. Furthermore, many of the existing implementations of ALGOL 68 are subset implementations, a fact resulting primarily from the

complexity of the language.

The TINMAN requirement that the translator minimize compiler costs appears to contradict the original language goal of efficiency. If most of the computing effort is to be in program execution (possibly in a production setting), then the critical factor is program execution time rather than program compilation time. The efficiency criterion established appears to indicate that such was the case. In that setting, then, extra compilation time could be used to optimize the object code produced, hence leading to more efficient programs.

The ALGOL 68 language provides adequate facilities for writing the compiler in ALGOL 68. The structure of translators and the certification of those translators with respect to diagnostic messages and compile-time checking is more of an implementation issue and an enforcement issue than a language design issue. These requirements should be imposed once a language has been selected to make sure that the translators behave similarly and that programs written for one environment will run in another. No such issues are explicitly treated in the ALGOL 68 report.

14. LANGUAGE STANDARDS, DEFINITION, AND CONTROL

This TINMAN section addresses itself primarily to the construction of a new language rather than to the evaluation of existing languages. It is thus hard to compare ALGOL 68 against these requirements.

ALGOL 68 must be viewed as being highly innovative in its use of language features, terminology, and related topics. It was designed with great care and the language is quite smooth considering the extent of the features.

The syntax and semantics of ALGOL 68 are defined unambiguously in the Revised Report on ALGOL 68. In fact, the

definition is such that many of the semantic notions are incorporated into the grammar. The grammar was developed by van Wijngaarden specifically for this language. Grammars of this two-level type are now called W-grammars.

User documentation has recently become available. Pagan's A Practical Guide to ALGOL 68 is keyed to the revised report and is extremely readable. A new edition of Lindsey and van der Meulen's Informal Introduction to ALGOL 68, which is somewhat more formal, is in press. An excellent ALGOL 68 tutorial by Tanenbaum appears in the June, 1976 issue of Computing Surveys.

# SECTION 3. LANGUAGE EVALUATION SUMMARY FOR ALGOL 68

HOL EVALUATION PROJECT:
LANGUAGE EVALUATION SUMMARY FOR ALGOL 68

A. DATA AND TYPES

A01 S
A02 P
A03 P
A04 N
A05 P
A06 P
A07 P

B. OPERATIONS

B01 S
B02 P
B03 S
B04 P
B05 S
B06 P
B07 S
B08 P
B09 N
B10 S
B11 N

C. EXPRESSIONS AND PARAMETERS

C01 N
C02 P
C03 S
C04 N
C05 P
C06 P
C07 P
C08 N
C09 P

D. VARIABLES, LITERALS, AND CONSTANTS

D01 S
D02 P
D03 S
D04 N
D05 S
D06 S

E. DEFINITION FACILITIES

E01 S
E02 S
E03 S
E04 S
E05 N
E06 P
E07 S
E08 N

F. SCOPES AND LIBRARIES

F01 S
F02 N
F03 S
F04 U
F05 U
F06 U
F07 S

G. CONTROL STRUCTURES

G01 P
G02 N
G03 P
G04 P
G05 P
G06 S
G07 P
G08 N

H. SYNTAX AND COMMENT CONVENTIONS

H01 P
H02 P
H03 S
H04 S
H05 P
H06 P
H07 P
H08 S
H09 N
H10 S

I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

I01 S
I02 P
I03 P
I04 N
I05 P
I06 N
I07 S

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCE

J01 U
J02 N
J03 N
J04 N
J05 N

K. PROGRAM ENVIRONMENT

K01 S
K02 N
K03 U
K04 U
K05 S

L. TRANSLATORS

L01 N
L02 N
L03 U
L04 S
L05 U
L06 U
L07 N
L08 S
L09 S

M. LANGUAGE DEFINITION, STANDARDS, AND CONTROL

M01 S
M02 S
M03 S
M04 N
M05 N
M06 N

Key: S = Satisfied
P = Partially Satisfied
N = Not Satisfied
U = Undetermined

# SECTION 4. DETAILED COMPARISON OF ALGOL 68 TO THE TINMAN

## 1. INTRODUCTION

This section outlines specific evaluations of each TINMAN requirement as they are (or are not) met by the language ALGOL 68.

Page number references when given refer to the basic reference document for the language: A. Van Wijngaarden, et.al., "Revised Report on the Algorithmic Language ALGOL 68," Acta Informatica, Volume 5, pages 1-236, 1975.

A. DATA AND TYPES

A1. Typed Language
Satisfied

It is possible to pass variable length arrays of a declared mode as parameters. Variables may be specified to be the union of two or more types: union (int,char) abc.

A2. Built-in Data Types
Partially Satisfied

No fixed point

A3. Precision Declaration
Partially Satisfied

Variables can be declared real, long real, long long real, short real, etc. to provide varying precision. The meaning of the modifiers is implementation-dependent. Some implementations of ALGOL 68 do not support this feature.

A4. Fixed Point
Not Satisfied

A new mode with operators can be defined to accomplish this.

A5. Enumeration Sense of Character Sets
Partially Satisfied

A mode can be defined to have the desired elements and desired ordering:

ASCII = [1:128] char = (skip, skip..."A","B",...)

A6. Arrays
Partially Satisfied

An array of variable size may be declared in a closed clause or a procedure. One can declare:

flex [n:n,p:1] int aa

or flex [1:0] real c

The dimensions of c become [1:3] upon execution of:

c:= (1.5,2.5,3.5)

The number of dimensions and the type of values remains static.

A7. Record Equivalence
Partially Satisfied

Single struct only, but flexible arrays and proc's may be part of a struct. Union Type may be used to achieve this objective.

B. OPERATIONS

B1. Assignment
Satisfied

B2. Equivalence Operator
Partially Satisfied

Standard prelude defines the modes which can be compared for equality, but the operator = (or eq) can be extended to compare objects of any mode, should new modes be introduced.

B3. Relationals Defined
Satisfied

B4. Arithmetic Operations
Partially Satisfied

÷ returns an integer value upon dividing two integers

/ returns a real permitting division of two reals, or of an integer by an integer, or of one by the other.

(ref. 10.2.3.3b, 10.2.3.4, 10.2.3.5)

B5. Truncation/Rounding
Satisfied

This is dependent upon the implementation's handling of fixed point overflow. The absence of fixed point facilities helps in meeting this condition.

B6. Boolean Operations
Partially Satisfied

Nor = >(a B) not built-in, but can be easily defined as:

op ≮ nor ≯= (bool a,b) bool : (a b/false/true)

No short-circuit; evaluations carried out (IBM 360 ALGOL 68C has ANDF, ORF).

B7. Scalar Operation
Satisfied

Also permits slicing of arrays

B8. Type Conversion
Partially Satisfied

Widening is a form of coercion - conversion from int to real, bits to bool, and bytes to [ ]char are done automatically when needed. No fixed point.

B9. Range Exceptions
Not Satisfied

No exception-handling for run-time truncation; however, run-time truncation should not occur. No range in ALGOL-68.

B10. I/O Operations
Satisfied

Control information - new page, new line, specific character position, backspace. Exceptions - end logical file, end physical file, value error, char error.

B11. Power Set Operators
Not Satisfied

No power sets of enumeration provided.

C. EXPRESSIONS AND PARAMETERS

C1. Side Effects
Not Satisfied

ALGOL 68 has collateral clauses in which expression order is expressly undefined. (Ref. 2.1.4.2e)

C2. Operand Hierarchy
Partially Satisfied

There are 10 levels of operator hierarchy, which may be changed by the programmer.

C3. Expressions Permitted
Satisfied

C4. Constant Expressions
Not Satisfied

No specification concerning compile-time evaluation of constant expressions. An implementation may do this.

C5. Parameter Rules
Partially Satisfied

Procedures may receive arrays where the size is unknown.

C6. Parameter Agreement
Partially Satisfied

Limited forms of widening, e.g. int to real are permitted.

C7. Formal Parameter Kinds
Partially Satisfied

No class for specifying control action when exceptions occur.

C8. Formal Parameter Specifications Optional
Not Satisfied

Note conflict with C6. Dimension is variable, but number of dimensions isn't. However, by using appropriate union definitions, generic procedures can be written.

C9. Variable Numbers of Parameters
Partially Satisfied

One can have a pointer to a linked list and pass the pointer as parameter or can have arrays of variable size, but, in general, a fixed number of arguments. Specific procedures, e.g. print, write, are present.

D. VARIABLES, LITERALS AND CONSTANTS

D1. Constant Value Identifiers
Satisfied

Use of = int K=15 is a constant

D2. Numeric Literals
Partially Satisfied

No specification for precision of real literals between object machines.

D3. Initialization
Satisfied

D4. Ranges and Stepsize
Not Satisfied

No subranges are built into the language.

D5. Variable Types
Satisfied

D6. Pointer Variables
Satisfied

Called ref variables, refer to objects of given type.

E. DEFINITION FACILITIES

E1. Type, Operator Definitions
Satisfied

New modes with operators; no encapsulation of modes with operators.

E2. Consistent Use
Satisfied

They may be used everywhere, but there are no built-in mechanisms for equal, read, print, etc. Existing operators may be extended for them.

E3. No Default Declarations
Satisfied

There is a standard prelude which explicitly makes a set of declarations and environment inquiries.

E4. Extend Existing Operators
Satisfied

E5. Type Definitions
Not Satisfied

No abstract types

E6. Data Definitions
Partially Satisfied

No power sets

E7. No Free Union
Satisfied

E8. Type Initialization and Allocation
Not Satisfied

F. SCOPES AND LIBRARIES

F1. Allocation and Access
Satisfied

F2. Limit Access Scope
Not Satisfied

No access limitations on mode declarations

F3. Static Scope Determination
Satisfied

F4. Libraries Available
Undetermined

F5. Library Contents
Undetermined

Implementation dependent - not ruled out by Reference.

F6. Libraries and Compools
Undetermined

F7. Machine Dependent Interface
Satisfied

The meaning of "special hardware" is unclear.

G. CONTROL STRUCTURES

G1. Control Structures
Partially Satisfied

Limited exception-handling, mostly confined to transput; no process control asynchronous interrupts.

G2. GOTO
Not Satisfied

Inadequate restrictions on target of the GOTO's.

G3. Conditional Control Structures
Partially Satisfied

The else part of an if...then is optional since fi brackets the if and eliminates the dangling else problem.

G4. Iteration Control
Partially Satisfied

Termination at loop beginning or via branch out of loop

G5. Recursive Routines
Partially Satisfied

Can define procedures within a recursive procedure and invoke them.

G6. Parallel Processing
Satisfied

Semaphores have operators down and up to permit mutual exclusion and synchronization.

G7. Exception Handling
Partially Satisfied

Limited number of transput exceptions

G8. Synchronization Primitives
Not Satisfied

Not specified in language; OS dependent

H. SYNTAX AND COMMENT CONVENTIONS

H1. Syntax Characteristics
Partially Satisfied

Keywords (Indicants) can be abbreviated in some cases, e.g., ▼ for down.

H2. No Syntax Extensions
Partially Satisfied

Can define new operator priorities, new operators, but no new keywords.

H3. Source Character Sets
Satisfied

H4. Literals and Identifiers
Satisfied

H5. Continued Lexicals Units
Partially Satisfied

No character for end-of-line in literal strings

H6. Keywords
Partially Satisfied

Implementation dependent, requirement holds in reference language - some implementations reserve about 90 words.

H7. Comment Convention
Partially Satisfied

Comments may begin with comment, co, #, or ¢. They must terminate with the same symbol.

H8. Unmatched Parentheses
Satisfied

H9. Uniform Referent
Not Satisfied

[ ] for subscripts; ( ) for functions

H10. Consistency of Meaning
Satisfied

I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

I1. Defaults in Program Logic
Satisfied

I2. Default Override
Partially Satisfied

No programmer default override

I3. Compile-time Parameters
Partially Satisfied

Environment inquiries are part of standard prelude.

I4. Object Environment Compile-time Parameters
Not Satisfied

No conditional compilation - can discuss word size, etc.

I5. Language Kernel
Partially Satisfied

Not a particularly small base - TINMAN requires a relatively large base.

I6. Translator Restrictions Defined
Not Satisfied

No translator limits defined

I7. Object Environment Restrictions
Satisfied

## J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCE

J1. Efficient Code
Undetermined

Unknown - heaps, recursion, other features may be inefficient.

J2. Optimization Invisible
Not Satisfied

Evaluation order may change the value of an expression.

J3. Access to Machine Language
Not Satisfied

No machine language code insertions - ALGOL 68 not a systems implementation language - effect can be achieved by manipulation of bits and bytes. Note: there are no compile time conditionals.

J4. Implementation Representation Tradeoffs
Not Satisfied

J5. Open/Closed Implementations
Not Satisfied

No macro-definition; all closed routines.

# CHAPTER 6
# LIS

The contents of this chapter of the report involve material relating to LIS.

Section 1. Evaluation of LIS Against Language Design Criteria

Section 2. Evaluation of LIS Against the TINMAN

Section 3. Language Evaluation Summary for LIS

Section 4. Detailed Comparison of LIS to the TINMAN

# SECTION 1. EVALUATION OF LIS AGAINST LANGUAGE DESIGN CRITERIA

## 1. GENERAL COMMENTS

LIS, or "Language for Implementation of Systems" , clearly is a language oriented to implementation of "systems" programs. It "...belongs to the same stream of thought as the languages PASCAL, SUE, and SIMULA" (page 1). If LIS can be characterized at all, it is by this primary attribute: its intended use as a system implementation language.

There are many clues in the LIS definition that (quite naturally, given LIS's direction) corroborate this. A few of them are worth noting:

- LIS has no built-in REAL numbers or variables, either floating point or fixed point. Thus, if LIS is to be used for "conventional" programming, the language would have to have such types added.
- The built in data structures of LIS are oriented to (i) sets of objects, (ii) arrays of objects (i.e. ordered sets), and (iii) "plexes" which are structured record-type objects.
- The control structure in LIS is rich--perhaps "richer" than would otherwise be appropriate to a language as simple as LIS. This, too, seems to bias the suitability fo LIS towards complex "logical operations" as opposed to numeric computation.
- The inter-program communication structures provided by LIS are also quite rich. Besides the conventional routines format there is provision for recursion (in LIS PROCEDURES), and for co-routines. Surprisingly, however, there are no synchronization primitives built-in for LIS. These are easily handled with existing features, though.

Thus, LIS seems an able language for handling a problem that involves (i) much logical operation, (ii) relatively complex data structures, and (iii) integer/character oriented data. For an operating system as it conventionally perceived, this seems to be just the right mix.

LIS is regarded as a marginally satisfactory "base language" which might be amenable to extension to meet the specific requirements of the TINMAN.

## 2. ABSTRACTION

The degree to which LIS supports the notion of abstraction can be measured by, perhaps equivalently, investigating facilities within LIS that provide the ability to "hide" information. Overall, LIS scores reasonably well in this area, for the following reasons:

1. LIS requires a programmer to give complete descriptions of data and symbols relevant to each program unit. Consequently, "details" that a programmer would like to localize to one or more related program elements can be isolated there.

2. LIS provides a relatively rich inter-unit control structure, making it possible to pass "contexts" to a serving sub-program without having to know all of the details of what is done to the information in that sub-program.

3. LIS provides a limited capability for abstraction of data.

4. The uniform reference feature hides the representation of objects which may be treated in different ways.

Thus, LIS makes it possible (although, in some cases only with some programmer difficulty) to hide details from a programmer at a higher level.

On the other hand, LIS's capabilities do not extend to the definition of abstract objects (data structures and operations between them) by which it would be possible to fully encapsulate a part of a program solution. Part of the reason for this is that LIS's facilities for extension do not permit new operators and/or extension of existing operators to newly defined data abstractions. Hence, in this situation a programmer would have to resort to the more conventional method of

providing operators by writing sub-programs which support them on a one-by-one basis.

3. MODULARITY

LIS provides good support for modularity. This is achieved through traditional scoping rules associated through block structuring and by parameter passing rules which require parameter specifications to include the declaration of IN, OUT, or both to designate their use. Except with the use of direct code insertions, program modules cannot gain access to the body of other modules. This concept extends to data objects as well, through the uniform reference facility.

The detriments to modularity include the ability to insert machine code and the ability to use global variables within inner blocks, unless those variables have been declared PRIVATE.

In general, though, LIS includes the facilities needed to decompose problems into modules and to keep modules isolated throughout program construction.

4. LINEAR FLOW OF CONTROL

The control structures of LIS are sufficiently rich that it is easy to write programs which exhibit linear flow of control within a module and which conform to the one-in, one-out rules of structured programming. Since LIS has no GOTO statement, the language imposes a discipline of linear control flow. The programmer must be careful in using the rich control structures so that no unnecessary control structures are added. There is a definite possibility that extensive nesting of IF clauses could lead to excessive use of the EXIT statement from inner clauses, thereby defeating the purpose of the control structures.

## 5. CONTROL OF SCOPE AND BINDING

LIS variables are bound to a type and have a scope of allocation determined at compile time. Variables may be declared PRIVATE to a block so that they will not be regarded as global (i.e., accessible) in contained blocks. The variables may be further bound to a specific object representation through the USE facility.

All LIS variables are declared in this way, so that LIS has all of the desirable properties of scoping associated with block-structured languages. In this way, the LIS scope rules assist in supporting the language design goals of abstraction and modularity.

## 6. READABILITY AND COMPREHENSIBILITY

LIS programs are generally readable by the knowledgeable LIS programmer. However, the cues to understanding are somewhat different from those which one might use to read a PASCAL program. With PASCAL, for example, one has the tendency to look at choices of data representation. Given the LIS approach to data independence, one must look at the program logic first and suppress knowledge about data representations. This process initially goes against the intuition of jointly studying program and data structures.

LIS provides good commenting facilities and ample freedom for use of mnemonic variable and type names. The keywords are relatively few and are generally meaningful.

As with other system implementation languages, though, the programmer has the ability to create highly convoluted programs by extensive use of direct code, by use of the SET data type to do bit manipulation, and by large amounts of nesting of control structures and PLEX definitions. Since it is possible to write an elegant program in even the worst

of languages and a poor program in the "best" languages, there is no way to guarantee readable programs. However, LIS seems to encourage good programming style more than most other languages.

7. PROCEDURES AND DATA

LIS satisfies the requirement for a procedure to be deterministic. As LIS is defined there is no way for a program in LIS to modify itself, except by machine language insertions over which a language compiler has no control.

8. LANGUAGE SIZE

The LIS language is "small" in terms of the number of different syntactic constructions it provides (including built-in data types and operators). For instance, the "syntax flow diagrams" for LIS number only about 30; each one, of course, involves a number of different syntactic elements. There are only some 150 productions in the BNF definition of LIS, making it comparable to PASCAL in size.

9. ANALYZABILITY

Given a syntactically legal LIS program, how difficult would it be to analyze (as in an automated program analysis tool other than a compiler)? The answer to this question for LIS appears to be "not difficult at all." Several features of LIS support this:

1. The cleanness of the input format (blank delimited, reserved words, clarity in lexical units, etc).

2. The simplicity of program written in LIS (although not necessarily their length).

3. The limited range of applications to which LIS, as it presently is defined, seems appropriate (namely, "systems programs").

4. The elimination of the troublesome GOTO and labelled statements.

Thus, an automated program analyzer would have an easy time with LIS codes.

## SECTION 2. EVALUATION OF LIS AGAINST THE TINMAN

## 1. OVERVIEW

LIS supports a number of the features which are important in the creation of high quality systems software. With respect to the TINMAN, its strengths lie in the requirement for full declarations, the support that it provides for low level representations of objects, the control structures of the language, and the ability to unite separately compiled program units.

Because it was designed as a system implementation language rather than as a general purpose programming language, some of the features desired by TINMAN are missing. In particular, there is no mechanism for fixed or floating point numbers, a weak mechanism for multi-dimensional arrays, and very limited input/output facilities (no input at all).

Thus, it would be very difficult to write, for example, a matrix multiplication routine in LIS. Similarly, it would be difficult to write a variety of data processing applications programs involving use of sequential or direct access files (or any form of input at all). The strength of LIS lies in its utility for the construction of compilers and operating systems and for the degree of discipline which is enforced concerning type checking, use of parameters, etc.

LIS is a marginal candidate as a base language for TINMAN or for a language design predicated upon the requirements of the TINMAN. Its shortcomings are too great and the language addresses too narrow a range of applications. However, LIS contributes several features to programming language design which it appears to solve more effectively than any other language of which the project team is aware.

In particular, the PLEX is especially elegant. This feature alone provides a record-handling mechanism, permits

alternative structures to be declared in the record structure, and yields a uniform referent notation, by permitting objects within a record to be static structures or actions, i.e., to have their value computed by a function at run-time.

Another nice feature of LIS is the encapsulation of machine dependencies. While the inclusion of direct code runs counter to most of the other requirements of TINMAN, LIS protects against improper use of direct code more effectively than almost any other language, with the possible exception of revised (January, 1977) Euclid. LIS provides two levels of data independence rather than the one level which is common in other languages.

A third good quality in LIS is the collection of control structures, which includes spawning of independent processes.

In summary, then, the designers of a new programming language which includes system implementation as an application should study LIS carefully and should consider using the LIS approach for record structures (PLEXes) and for direct code insertions, regardless of the base language used for the language design.

## 2. DATA TYPES

LIS is a strongly typed language, requiring all procedures and data objects to be completely declared prior to compilation, and forbidding these types from being altered at run time (with the exception of machine code insertions).* While LIS supports integers, arrays, enumeration types, power sets of enumeration types, Booleans, and characters, it does not provide for

*There are numerous instances where the intent of the language design can be subverted by the insertion of direct code, e.g., to work with data as both integers and strings. It is assumed that the reader is aware of some of the potential problems posed by direct code. Hence, statements made about language characteristics will hereafter apply to the high-level language, with the understanding that any restrictions may be circumvented by use of machine code.

non-integer arithmetic. As noted above, LIS provides the concept of a PLEX to accomplish the requirements of a record-handling capability. The PLEX must be viewed as one of the most innovative contributions made by LIS. CASE statements embedded in the PLEX declaration permit records to have safely described alternative structures.

## 3. OPERATIONS

Assignment and reference operations are handled in the expected manner, with type checking in assignments. The relational operators are defined between any two simple expressions, where those simple expressions may designate, for example, an array or a single data element. The desired logical operators are present, except that XOR replaces nor. Scalar operations on arrays are nicely handled in LIS.

LIS does not permit side effects in functions, i.e., no parameter to a function may be altered, so that an implementor could implement short circuiting of logical operations without affecting the correctness of programs. (However, this feature is not specified in the language definition.) The abolition of side effects from functions, also followed in Euclid, GYPSY, and PLAIN, is regarded as essential to the fulfillment of the TINMAN objectives.

LIS fails to fulfill several of the objectives in this area, though. The absence of non-integer arithmetic means that several kinds of desirable operations are not present. While numeric subranges are handled, they are treated as separate types in the language declarations; the language reference manual is not explicit about checking of arithmetic expressions and operations involving objects of subrange type. Since subranges apply only to integer and enumeration types, the subrange objects use the same operations as those types.

LIS does not support input/output very effectively. There are no input facilities and very limited output facilities. Finally, the built-in operations on power sets are quite limited, although the existence of SET as a basic types, combined with function definition capability, makes it easy to provide operations (buy not binary operators) for union, intersection, difference, and complement.

4. EXPRESSIONS AND PARAMETERS

LIS is quite strong with respect to this set of TINMAN requirements. The number of levels of operator hierarchy is only four, the language is consistent with respect to the use of expressions, and the rules concerning parameter passing to functions eliminate side effects, so that the order of expression evaluation is not affected by optimization. LIS contains a good facility for the declaration and use of constants, although some constants must be evaluated at run time upon block entry. Type checking is performed between the formal and actual parameters on procedure calls.

LIS fails to meet some of the TINMAN requirements related to parameters, though. Parameters to subprograms may be of type value, result, or value-result. This technique, however, may not be used to pass the name of a subprogram as a parameter. LIS, instead, has a connector type for that purpose. There is no exception-handling mechanism in LIS and hence no parameter class for exceptions. LIS does not support generic functions (C8), nor are variable length parameter lists permissible.

5. VARIABLES, LITERALS, AND CONSTANTS

LIS provides the ability to associate constant values with different data types and to initialize variables of any type, although such initialization is not required. Since fixed point variables are not present in LIS, there is no provision

for specifying the range and step size of such variables.

LIS contains the REF mechanism as a pointer, which can be used to build data with shared and/or recursive substructures. The ROW facility provides a simplified way of indirectly pointing at an array.

## 6. DEFINITION FACILITIES

LIS does not provide for a data definition facility commensurate with the concept of abstract types. A programmer may define sets, plexes, and arrays, which are built upon simpler types, and may define the value of an object in a plex to be computed as an action, but there is no encapsulation of the object and its operators. It should be noted, though, that the ACTION mechanism comes close to what is desired. If one has declared

```
TYPE PERSON =
     PLEX
        BIRTH:  CONSTANT INTEGER;
          .
          .
        GETAGE: ACTION (AGE: OUT INTEGER);
          .
          .
     END;
```

then one may declare

```
JOHN: PERSON
```

and the use of JOHN.GETAGE provides some of the desired information hiding. However, there is no extension of existing operators to new types, and no facility for introducing new binary operators.

The definition facility, however, provides several ways for associating data objects with their defined type at compile time, including enumeration, array, and plex declarations. These have initialization procedures, but not finalization procedures.

## 7. SCOPES AND LIBRARIES

LIS permits variables to be declared PRIVATE so as to distinguish the scope of access from the scope of allocation. Private variables may not be accessed by blocks subordinate to the block in which the private variable is declared. Similarly, restrictions are placed on the visibility of data segments. The scoping rules of LIS are explicit and easily determined from the program structure at compile-time, as with other block-structured languages.

LIS provides a powerful facility for separate compilations, based on the presumed existence of libraries to store procedures and data segments. Furthermore, the implementation part of LIS envisions the inclusion of program segments written in languages other than LIS. These implementation parts may be included in LIS libraries along with procedure and data segments written in LIS itself.

LIS, however, does not include the facilities for machine independent interfaces to machine dependent capabilities, except in the most general sense, i.e., that machine language routines can be included.

## 8. CONTROL STRUCTURES

The LIS language has a powerful set of control structures, including LOOP-REPEAT, WHILE-LOOP-REPEAT, UNTIL-LOOP-REPEAT, a FOR statement, a FOR EACH statement, a CASE statement, an IF-THEN-ELSE, and subroutine calls and returns, including recursion. Subprograms may be invoked thorugh normal calls or through the use of connectors or references to objects in a plex where the object designates an action. Co-routines are also included in LIS.

With respect to the TINMAN requirements, the LIS syntax could be judged to have some shortcomings. However, there could easily be some debate as to whether failure to meet these

requirements are flaws in LIS of flaws in the TINMAN. For example, LIS permits nesting or recursive routines. Also, there is no exception handling mechanism. Finally, there is no real-time interrupt facility. Of course, TINMAN also requires a GOTO statement, which is not present in LIS.

On the whole, however, LIS control structures are among the best of all programming languages and give the programmer an excellent choice of control structures without any needless restrictions. The absence of the GOTO statement does not detract from programmability in any way. Specific characteristics of LIS which should be observed by the designers of a language to fulfill the TINMAN (or its successors) include the FOR statement, in which the control variable is treated as a locally declared variable, and the PROCESS mechanism, which permits the spawning of asynchronous processes. Parallelism is achieved by declaring a PLEX SHARED, in which WAIT and SIGNAL are attributes declared as ACTIONs, along with an integer semaphore. Thus, an instance of this plex may be associated with every shareable object in a system, thereby providing the facility for mutual exclusion. This capability is one of the more elegant features of LIS.

## 9. SYNTAX AND COMMENT CONVENTIONS

LIS programs are highly readable once the barrier of some of the new LIS constructs is overcome. LIS presents some features which are innovative in programming languages, and it takes a little experience before one becomes comfortable with these features. However, the complexity is considerably less than that of ALGOL 68. The readability of LIS programs is enhanced by the "prettyprint" processing of the translator.

The uniform reference notation is achieved in LIS (a major achievement), but has the disadvantage of detracting from program comprehensibility beyond a certain point. At a "high-level" reading of the program, uniform referents are

highly desirable, but when it is necessary to look at the lower levels of program construction, the loss of redundancy (compared with PASCAL) provided by the varying reference notations can be disconcerting. It is not immediately clear that the advantages of uniform referents (data independence) outweigh the disadvantages until one has considerable experience with their use. (This difficulty can be likened in some ways to the transition between file systems and data base management systems, in which many people experienced with file system were uncomfortable without knowing the representation of their data on secondary storage.)

The syntax is quite clean and free of duplications, except for the annoyance of some similarly constructed statements for assignments, initializations, and parameter results. = is used for both TYPE declarations and as a relational operator, initializations use ==, and input/output parameters use :=:. The parameter symbols must correspond with the IN and OUT notations of the formal parameter declarations; the use of two different notations, one in the procedure call and one in the procedure declarations, seem unnatural.

One cannot write self modifying programs in LIS. Everything is identifiable at compilation time. The syntax of the language cannot be changed and the operator hierarchies are immutable.

Even the break characters are present in LIS. The underscore character (_) may be used in the formation of names, e.g. IBM_370, and strings may be split, e.g., "HELLO" "THERE" is equivalent to "HELLOTHERE". There is no break character for integers, though.

Again, LIS rates very high on issues of language syntax and comment conventions.

## 10. DEFAULTS, CONDITIONAL COMPILATION, AND RESTRICTIONS

LIS does not provide defaults in program logic. Although there are defaults in object representation, the programmer has the opportunity to override those defaults through use of the USE statement.

The LIS language base is close to minimal, although there are several instances where there is more than one way to accomplish the same objective. For example, the FOR statement is inherently redundant, since it can be handled with a WHILE-LOOP. Also, the ROW feature could be adequately handled with the REF capability. There are a couple of other similar instances along this line, but they are relatively minor. LIS is well designed and hence has few unnecessary features; those duplications are given simply to make certain common operations easier.

## 11. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

The LIS programmer has access to direct code and to machine dependencies such as register allocation. This feature provides the ability to specify the object representation of composite data structures where desired. However, there is no "macro" facility in LIS which would permit both open and closed subroutines. Procedures, functions, and actions are all closed.

The language is moderate in size so that it should be possible to develop a reasonably efficient compiler. However, there are a few aspects of the language related to separate compilations and linkage of high-level data objects with their representations where some inefficiency may result. In particular, a programmer may specify a tightly packed object representation which would require the system to perform time-consuming packing and unpacking operations with each reference and each assignment to a data object.

12. PROGRAM ENVIRONMENT

LIS is intended to run in an operating environment which is capable of supporting asynchronous processes, subprogram libraries, and several other operating system facilities. To some extent, then, the requirement in TINMAN for parallel processing and asynchronous interrupt handling requires such an operating system. LIS features which necessitate a sophisticated operating system include the USE facility, the separate compilation ability, and the PROCESS declaration.

The language declaration does not make mention of the other tools available to the LIS programmer, but it is assumed that suitable text preparation, debugging, and testing facilities would be made available, since LIS would be an appropriate language for writing such programs.

13. TRANSLATORS

No standards have been established for LIS translators. To the knowledge of the project team, there is only one implementation of LIS. This implementation and the associated reference documents have been changing over the past several years, in keeping with typical language design efforts in which changes are made as experience is gained in use and implementation of the language. In that way, the existing compiler for the CII machine implements the language with no subset or superset implementation. It would be possible to write the LIS compiler in LIS.

Other aspects of translator characteristics are unknown. Because the project team is not informed as to the precise status of LIS translators, all of the evaluation criteria in this category have been left as undetermined. (This scoring may have detracted from the final relative score of LIS.)

## 14. LANGUAGE DEFINITION, STANDARDS AND CONTROL

LIS is innovative in several respects, so that it cannot be said that LIS is composed of proven features from other languages. However, the construction of a compiler for LIS demonstrates that the features of the language are well within the state of the art. Modifications are being made to the language and engineered into the translator.

As of yet, there is no formal definition for LIS along the lines of the W-grammar of ALGOL 68 or the proof rules which have been planned for Euclid. There were several instances within the language reference manual where the description was so terse or incomplete that the project team had to attempt to infer the intent of the language, with a lesser degree of certainty than was achieved for ALGOL 60 and ALGOL 68.

The language has been frozen recently, so it is possible that additional documentation will be forthcoming. We assume that some of this documentation is already available in French, but we have not had access to it.

Implementation and control of the language has been maintained until now by Jean Ichbian and CII. At this time, there appears to be only a single version of the source language so that another support agent could be appointed to manage the construction and validation of translators conforming to that language.

# SECTION 3. LANGUAGE EVALUATION SUMMARY FOR LIS

HOL EVALUATION PROJECT:
LANGUAGE EVALUATION SUMMARY FOR LIS

A. DATA AND TYPES

| | |
|---|---|
| A01 | S |
| A02 | N |
| A03 | N |
| A04 | N |
| A05 | P |
| A06 | N |
| A07 | S |

B. OPERATIONS

| | |
|---|---|
| B01 | S |
| B02 | N |
| B03 | S |
| B04 | S |
| B05 | N |
| B06 | P |
| B07 | S |
| B08 | N |
| B09 | N |
| B10 | N |
| B11 | S |

C. EXPRESSIONS AND PARAMETERS

| | |
|---|---|
| C01 | S |
| C02 | S |
| C03 | S |
| C04 | P |
| C05 | P |
| C06 | S |
| C07 | P |
| C08 | N |
| C09 | N |

D. VARIABLES, LITERALS, AND CONSTANTS

| | |
|---|---|
| D01 | S |
| D02 | S |
| D03 | S |
| D04 | N |
| D05 | S |
| D06 | S |

E. DEFINITION FACILITIES

| | |
|---|---|
| E01 | P |
| E02 | S |
| E03 | S |
| E04 | N |
| E05 | N |
| E06 | P |
| E07 | N |
| E08 | N |

F. SCOPES AND LIBRARIES

| | |
|---|---|
| F01 | P |
| F02 | P |
| F03 | S |
| F04 | U |
| F05 | U |
| F06 | P |
| F07 | S |

G. CONTROL STRUCTURES

| | |
|---|---|
| G01 | P |
| G02 | N |
| G03 | P |
| G04 | P |
| G05 | P |
| G06 | P |
| G07 | N |
| G08 | N |

H. SYNTAX AND COMMENT CONVENTIONS

| | |
|---|---|
| H01 | S |
| H02 | S |
| H03 | N |
| H04 | P |
| H05 | P |
| H06 | S |
| H07 | S |
| H08 | S |
| H09 | S |
| H10 | P |

I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

| | |
|---|---|
| I01 | S |
| I02 | S |
| I03 | N |
| I04 | U |
| I05 | S |
| I06 | U |
| I07 | U |

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCE

| | |
|---|---|
| J01 | U |
| J02 | U |
| J03 | S |
| J04 | P |
| J05 | N |

K. PROGRAM ENVIRONMENT

| | |
|---|---|
| K01 | S |
| K02 | S |
| K03 | U |
| K04 | U |
| K05 | S |

L. TRANSLATORS

| | |
|---|---|
| L01 | U |
| L02 | U |
| L03 | U |
| L04 | U |
| L05 | U |
| L06 | U |
| L07 | U |
| L08 | U |
| L09 | U |

M. LANGUAGE DEFINITION, STANDARDS AND CONTROL

| | |
|---|---|
| M01 | P |
| M02 | S |
| M03 | P |
| M04 | U |
| M05 | U |
| M06 | U |

Key: S = Satisfied
P = Partially Satisfied
N = Not Satisfied
U = Undetermined

# SECTION 4. DETAILED COMPARISON OF LIS TO THE TINMAN

## 1. INTRODUCTION

This section outlines specific evaluations of each TINMAN requirement as they are (or are not) met by the language LIS.

Page number references are stated for The System Implementation Language LIS, C.I.I., Document No. 4549 EL/EN, January 1976.

A. DATA AND TYPES

A1: Typed Language
Satisfied

LIS is fully typed (page 9 ff). Each entity must be declared and each such declaration must include a type specification. This is true both for built-in types and user-defined types (page 66 ff).

A2: Built-in Data Types
Not Satisfied

LIS provides predefined types for BOOLEAN, INTEGER, and CHARACTER (page 13 ff). In BOOLEAN, TRUE compares greater than FALSE. Fixed-point and REAL data types are not pre-defined. Arrays, sets, and record-structure data types are available. LIS's plexes are "records" with generalized and variant structure.

A3: Precision Declaration
Not Satisfied

Because neither REAL nor Fixed-point data declarations are predefined, correspondingly there is no precision specification possible. Appropriate type definitions are possible, however.

A4: Fixed Point
Not Satisfied

As per comments for A3, there exists no facility to indicate Fixed-point range and scale. Appropriate type definitions are possible, however.

A5: Enumeration Sense of Character Sets
Partially Satisfied

The predefined type CHARACTER in LIS is a symbolic type for which the character values are ordered according to the EBCDIC codes (page 13). USASCII is not mentioned.

A6: Arrays
Not Satisfied

The LIS language only has the ARRAY data type, but no predefined mechanism for n-dimensioned arrays, n < 1. Note, however, that 2- and higher-dimensioned arrays can be implemented by using an ARRAY...OF...ARRAY structure, although the accessing mechanism that results is somewhat clumsy.

A7: Record Equivalence
Satisfied

LIS's capabilities to define PLEXes makes it possible to imply record equivalence. Both whole records and/or parts of records can be equivalenced in this manner by using the CASE construction within a PLEX declaration. (Page 15 gives an example.)

B. OPERATORS

B1: Assignment
Satisfied

LIS provides for appropriate conversions between all built-in types (page 31).

B2: Equivalence Operator
Not Satisfied

Equivalence between any two entities of the same type is defined unambiguously in LIS (page 32); but this capability is not extended to differing types.

B3: Relationals Defined
Satisfied

The required (standard) relational operators are all pre-defined in LIS (page 30).

B4: Arithmetic Operations
Satisfied

Although the standard arithmetic operations are defined in LIS, because of the limited number of pre-defined types the language makes no provision for floating point operations.

B5: Truncation/Rounding
Not Satisfied

LIS, except by user extension, does not support control of truncation and rounding in arithmetic operations.

B6: Boolean Operations
Partially Satisfied

LIS includes the XOR operation rather than the NOR operation (page 30). The "short circuit" mode is not permitted, but could be implemented without difficulty.

B7: Scalar Operations
Satisfied

Scalar operations in LIS extend to composite data structures (page 38).

B8: Type Conversion
Not Satisfied

While LIS meets this requirement, the set of data structures predefined in the language makes the issue a very minor one. LIS has only INTEGER and CHARACTER types and so type conversions are not defined in a manner such as that required by the TINMAN.

B9: Range Exceptions
Not Satisfied

LIS has no provision for exception condition processing.

B10: I/O Operations
Not Satisfied

LIS has no input/output facility except that provided by symbolic output (page 50).

B11: Power Set Operations
Satisfied

C. EXPRESSIONS AND PARAMETERS

C1: Side Effects
Satisfied

LIS specifically indicates a left-to-right scan (page 31) for operations of the same precedence, but does not imply a left-to-right scan for all operations. Side effects, though, cannot occur.

C2: Operand Hierarchy
Satisfied

LIS predefined operators have a predefined hierarchy.

C3: Expressions Permitted
Satisfied

Expressions can be tolerated anywhere a simple variable can appear.

C4: Constant Expressions
Partially Satisfied

LIS does not indicate (one way or the other) how a constant expression is processed except to reassure the user that constants are constant. The CONSTANT attribute can be applied to any identifier. Not all CONSTANT values can be evaluated at compile time.

C5: Parameter Rules
Partially Satisfied

Parameter definition and use rules for LIS are consistent and unambiguous, but there is no provision for exception handling.

C6: Parameter Agreement
Satisfied

C7: Formal Parameter Kinds
Partially Satisfied

LIS provides for three kinds of formal parameters: read-only, read-write, write-only. The read-only parameter mode is the same as "call by value"; the read-write parameter mode is the same as "call by reference-address." The connector type is used to accomplish passing of procedure names as parameters.

C8: Formal Parameter Specifications Optional
Not Satisfied

LIS makes no provision for inferring parameter types by referencing an invocation.

C9: Variable Numbers of Parameters
Not Satisfied

Although LIS requires that the minimum number of actual parameters required be present, it makes no statement about what happens where an invocation has more than the minimum number of parameters (page 47).

D. VARIABLES, LITERALS AND CONSTANTS

D1: Constant Value Identifiers
Satisfied

In LIS's declaration structure there is explicit provision for initialization (page 9).

D2: Numeric Literals
Satisfied

D3: Initialization
Satisfied

D4: Ranges and Stepsize
Not Satisfied

LIS, by not explicitly providing for fixed-point data types, thereby makes no provision for range and stepsize specifications.

D5: Variable Types
Satisfied

E. DEFINITION FACILITIES

E1: User Type, Operator Definitions
Partially Satisfied

LIS provides for "implementation parts" (page 64 ff) which can define storage management properties for data types defined by the user. The orientation of this facility fo LIS is, however, specifically toward bit/byte/word/double-word manipulations of memory layouts. The language definition does not permit the addition of new operators (or operator symbols).

E2: Consistent Use
Satisfied

For types that can be defined, the usage rules in LIS are consistent. This results from the fact that only new data types, and not new operators, can be defined.

E3: No Default Declarations
Satisfied

E4: Extend Existing Operators
Not Satisfied

LIS provides no facility for extending the meaning of any of the existing operators, nor for defining new operators.

E5: Type Definitions
Not Satisfied

There is no abstract type definition in LIS.

E6: Data Definitions
Partially Satisfied

All of the operations mentioned in the TINMAN except discriminated union are possible.

E7: No Free Union
Not Satisfied

Free Unions can exist in LIS (page 12).

E8: Type Initialization and Allocation
Not Satisfied

LIS gives no facility for defining the processing of new types and thus has no facility for intialization and/or allocation.

F. SCOPES AND LIBRARIES

F1: Allocation and Access
Partially Satisfied

LIS provides for inclusion of one entity in another and defines the scope rules for inner-defined quantities.

F2: Limit Access Scope
Partially Satisfied

LIS provides no general capability for limiting access. PRIVATE variables give a limited capability to control the scope of access to a variable.

F3: Static Scope Determination
Satisfied

LIS, being a well-structured and fully typed language with a requirement that all entities be fully declared, intrinsically accomplishes deterministic scoping.

F4: Libraries Available
Undetermined

LIS makes no mention of a program library facility; however, there are provisions for separate compilation which imply such a library.

F5: Library Contents
Undetermined

LIS includes no facility for accessing a compile-time library.

F6: Libraries and Compools
Partially Satisfied

LIS includes data segments which appear to resemble compools. They may be used in a general program structure.

F7: Machine Dependent Interface
Satisfied

LIS makes it possible to incorporate in-line sequences of machine language text (page 73). The layout used, and the associated rules strongly resemble System 360/370 coding conventions and register use rules.

G. CONTROL STRUCTURES

G1: Control Sturctures
Partially Satisfied

LIS provides for the standard structured porgramming control structures, with the addition of an "ELSE IF" element in the selector construct. LIS supports co-routine calls (page 47). LIS has no explicit facility for exception handling, or asynchronous interrupt handling. Note that co-routines could be used to implement the required synchronization primitives and parallel processing relatively easily.

G2: GOTO
Not Satisfied

LIS has no labelling facility and no GOTO statement.

G3: Conditional Control Structures
Partially Satisfied

LIS provides for Boolean condition selection and includes a CASE statement that provides a computed choice.

G4: Iteration Control
Partially Satisfied

LIS provides for termination anywhere in a loop (page 45), provides for subtyping of a value (either symbolic or literal) in a CASE-like statement (page 42), and, by default, assures top-entry to iteration since there is no capability for labelling statements. Control variables in LIS are handled properly.

G5: Recursive Routines
Partially Satisfied

LIS provides for recursive procedures (page 3), but it is also possible to define a procedure within a recursive procedure.

G6: Parallel Processing
Partially Satisfied

LIS provides the capability to allocate resources (FREE and ALLOCATE keywords, page 49) relating to PLEX definitions. LIS does not provide for explicit control of parallel processing.

G7: Exception Handling
Not Satisfied

LIS, in the language definition, makes no mention of exception handling, although it seems to be envisioned by Ichbiah.

G8: Synchronization Primitives
Not Satisfied

LIS has no built-in synchronization primitives. However, because LIS has co-routines and plexes with actions, these could be implemented rather easily.

## H. SYNTAX AND COMMENT CONVENTIONS

H1: Syntax Characteristics
Satisfied

The "clarity and comprehensibility" of LIS, a judgemental issue, is helped on two features: the capability to reparagraph the source text, and the capability for commenting.

H2: No Syntax Extensions
Satisfied

LIS provides for a fixed syntax.

H3: Source Character Set
Not Satisfied

LIS programs are defined in terms of a subset of EBCDIC (page 6), while the CHARACTER Mode operates on the full EBCDIC character set (page 13).

H4: Literals and Identifiers
Partially Satisfied

There is a break facility for strings. The underscore can be used in names. There is no break character in numbers.

H5: Continued Lexicals Units
Partially Satisfied

LIS requires fully juxtaposed lexical tokens. There is no End-of-Line Symbol, however.

H6: Keywords
Satisfied

LIS has 82 reserved keywords.

H7: Comment Conventions
Satisfied

LIS's comment convention (page 8) has easy to use and understand processing rules.

H8: Unmatched Parentheses
Satisfied

H9: Uniform Referent
Satisfied

H10: Consistency of Meaning
Partially Satisfied

LIS provides no capability to redefine the meaning of a language-defined symbol, but "=" is used both in type definitions and as a relational operator.

I. DEFAULTS, CONDITIONAL COMPILAITON AND LANGUAGE RESTRICTIONS

I1. Defaults in Program Logic
Satisfied

LIS, which requires full specification on the part of the programmer, supports the property wanted by the TINMAN.

I2. Default Override
Satisfied

LIS has no defaults.

I3: Compile-time Parameters
Not Satisfied

I4: Object Environment Compile-time Parameters
Undetermined

I5· Language Kernel
Satisfied

LIS in its entirety is small enough to be considered as its own kernel, although there is some redundancy.

I6: Translator Restrictions Defined
Undetermined

I7: Object Environment Restrictions
Undetermined

J. EFFICIENT OBJECT REPRESENTATION AND MACHINE DEPENDENCE

J1. Efficient Code
Undetermined

J2. Optimization Invisible
Undetermined

J3. Access to Machine Language
Satisfied

This is permitted in LIS (page 73 ff).

J4. Implementation Representation Tradeoffs
Partially Satisfied

LIS provides a limited facility for specifying machine layouts (page 65 ff), but refers only to bit/byte/word/double-word layouts in memory. This facility appears oriented to providing protection against storage reference mistakes due to machine changes.

J5. Open/Closed Implementations
Not Satisfied

The programmer in LIS has no control over the form of implementation of a subroutine/subprogram.

K. PROGRAM ENVIRONMENT

K1. Operating System Not Required
Satisfied

K2. Program Assembly
Satisfied

LIS's description includes the possibility of separately compiled modules.

K3. Software Development Tools
Undetermined

K4. Translator Options
Undetermined

K5. Assertion Capability
Satisfied

LIS can be made to provide for assertions treated as comments.

## L. TRANSLATORS

All of the TINMAN issues in regard to the translator for LIS are "undetermined", primarily for lack of detailed information about the LIS translator.

M. LANGUAGE DEFINITION, STANDARDS AND CONTROL

M1. Existing Features
Partially Satisfied

LIS contains a number of unique language features, including the PLEX implementation specifications, and attribute definitions, but all of these seem implementable in a straightforward way.

M2. Language Subsets
Satisfied

The LIS manual provides verbal explanations of the semantics of each LIS construct for which the interpretation could be different from the common assumptions. However, the description is not given in a rigorous semantic description format.

M3. Language Documentation
Partially Satisfied

To the extent that the LIS manual represents a tutorial, this requirement is satisfied.

M4. Control Agent
Undetermined

M5. Support Agent
Undetermined

M6. Library Standards
Undetermined

# CHAPTER 7
# CS-4

The contents of this chapter of the report involve material relating to CS-4.

Section 1. Evaluation of CS-4 Against Language Design Criteria

Section 2. Evaluation of CS-4 Against the TINMAN

Section 3. Language Evaluation Summary for CS-4

Section 4. Detailed Comparison of CS-4 to the TINMAN

# SECTION 1. EVALUATION OF CS-4 AGAINST LANGUAGE DESIGN CRITERIA

## 1. INTRODUCTION

This section examines the extent to which CS-4 satisfies the criteria for a "good" higher-order language as presented in "Criteria for Evaluation of High-Order Languages." CS-4 is seen to fulfill the needs moderately well. However, there are certain weaknesses in most of the areas, caused by the size of the language, the number of inconsistencies in the syntax and semantics, and the number of untested language features. CS-4 is marginally suitable for consideration as a base language for the TINMAN, but is a much weaker candidate than is ALGOL 68 or PASCAL.

## 2. ABSTRACTION

CS-4 provides features for procedural abstraction through functions and procedures. A function is a procedure with a result specification. Procedures accept a variety of parameters: these parameters fall into major classes: call by value, call by reference, and call by name. Call by value is identical to the PASCAL notion, call by reference also follows PASCAL and ALGOL 68, call by name follows ALGOL 60. Each of these three classes is further distinguished by designating whether a parameter is input, output, or both. An additional parameterization facility is provided by the ATTRIBUTE concept, which permits exceptions to be passed.

There are some problems associated with permitting both call by name and call by reference. Call by name is intended to be used mainly for passing the name of a procedure as a parameter, but is not restricted in any sense. Thus, one may pass algebraic expressions (Jensen's device) and (it would appear) a constant by name just as well. Ledgard has shown (Computing Surveys, Sept., 1971, pp. 126-7) some of the potential for confusion when both call by reference and call by name are permitted.

CS-4 also has a facility for data abstraction. However, the parameter passing rules for data abstractions are different from those of procedures. Some of the better features of the CS-4 data abstraction facility are the ability to designate the names which are accessible outside the mode definition (CAPABILITY) and the protection given to local names. Furthermore, the set of standard mode operations is useful and provides a degree of equivalence among data types. The ability to specify finalization (TERM) actions is unique to CS-4. On the other hand, though, one can object to the facility for mode generators incorporated within CS-4. One of the legal parameter types for a mode declaration is a mode itself, i.e., a mode is treated as a mode. This presents some implementation difficulties, defers the binding of modes of objects, detracts from the clarity of programs, and seems to run counter to the language philosophy of strict type checking.

Other suitable abstraction facilities are provided through a sizeable number of built-in data types, including integers, reals, booleans, arrays, records, strings, sets, union types, complex numbers, vectors, matrices, fractions, and enumeration types (STATUS). All of these built-in types have a set of associated operations. This large number of built-in types is a major contributor to the size of the CS-4 language, but makes the language suitable for a broad set of applications.

3. MODULARITY

CS-4 is a block structured language in which variables may be passed between blocks. A procedure with local declarations is considered a block for this purpose. A good deal of modularity is presented by the block structure and the parameter passing mechanism. Furthermore, CS-4 supports separate compilation of program modules and contains ACCESS and RENAME facilities to use desired names or eliminate conflicts between

names in separately compiled modules.

There are, though, some ways to subvert the intended modularity. One is through the use of global variables, which may be used within contained blocks. Access to names contained within a block may, as usual, be kept invisible to external blocks. Also, there is no control over side-effects in functions, as there is in such languages as Euclid and Gypsy. CS-4 leaves the order of expression evaluation undefined, so that it is possible to subvert the intent of modularity by having a module (that which implements the function) affect the computational result of another module by side effect.

In general, though, one can achieve modularity in CS-4 as well as with other modern programming languages, e.g., PASCAL. It is simply necessary to be careful in module construction and in the use of global variables and procedures.

## 4. LINEAR FLOW OF CONTROL

CS-4 has a number of different control structures which are intended to eliminate the need for unrestrained transfers of control. In addition to the procedure invocation, there is an IF statement, a CASE statement (including OTHERWISE), and the usual FOR, REPEAT, and WHILE constructions. The FOR statement can assure that the control variable is local to the scope of the loop. The GO TO statement prohibits transfers to inner blocks, but permits transfer out of several nested levels with a single command, contrary to the strictures of good programming style. A variant of the GO TO is the ABORT TO.

CS-4 also has an exception-handling mechanism which can be used to provide alternate flow of control in the event of some condition arising, possibly a user-defined condition. The SIGNAL statement (or the automatic occurrence of an exception) causes a procedure to be invoked and permits parameters to be passed. The SIGNAL may be passed to successive levels of

handlers with a RESIGNAL statement, a construct which is unknown and untested in other languages. It would appear that there is some danger of setting up a completely alternative program flow through exceptions and parameters which is completely different from the visual program structure. Once again, there are adequate facilities for writing a good program, but too many additional facilities which may lead to the construction of poor programs.

5. CONTROL OF SCOPE AND BINDING

CS-4 is similar to ALGOL 60 in this respect, in that the language has a block structure. The differences are that CS-4 has STATIC allocation (somewhat similar to ALGOL 60 own) and SHARED allocation for multiple processes, which can be protected or unprotected. SHARED allocation is somewhat like FORTRAN COMMON in that objects may be shared among modules with relatively little control; this can be used to destroy the intent of modularity, but is seen as being relatively useful for efficiency. (CS-4 programmers should be encouraged to use structured data and pass such objects as parameters by reference; a single parameter produces relatively little overhead.)

With the exception of SHARED objects, the scope of variables is as in ALGOL and PASCAL, with the block structure determining the scope of access. The only way to prevent inner blocks from accessing names declared in other containing blocks is by renaming those variables in the inner blocks.

CS-4 sets up a mechanism intended to provide a rational approach to scope and binding, then turns around and gives the programmer the ability to override that mechanism with the SHARED feature. In addition, the CS-4 programmer has the ability to make machine code insertions in programs, giving the ability to circumvent virtually all of the intended restrictions.

## 6. READABILITY AND COMPREHENSIBILITY

CS-4 is a large language by any measure. It is virtually devoid of defaults, so that even the most obvious attributes must be declared. For example, one cannot simply declare an integer (with range determined by implementation and undefined initial value), but must explicitly include a range and initialization. These requirements lead to a rather wordy program, which offers certain advantages and disadvantages with respect to readability. Some of the benefits in CS-4 are the ability to associate KEYWORDS, i.e., formal parameters, with actual parameters in a procedure call, a good commenting facility, no arbitrary restrictions on identifier length, and powerful abstraction facilities. It is clear that one can write a readable, if somewhat verbose, program in CS-4.

With respect to comprehensibility, though, CS-4 seems to rate somewhat lower. The complexity of the language is such that few people will completely understand all of its mechanisms. There are so many inconsistencies, such as the parameter passing rules dependent on what is being declared (procedure or mode). There are a vast number of built-in exception names. There are nearly 300 reserved words, in three different flavors, some of which can be overridden. There is a long list of built-in operators for each built-in type. Some of the syntactic constructs do not seem particularly logical, especially those in procedure headings. It is the rare programmer who will become fully conversant with the language to the extent that most programmers are comfortable with languages such as ALGOL 60, FORTRAN, or even COBOL.

The composition of correct CS-4 programs seems to require a great deal of attention to detail and memorization of a large number of syntactic forms. The number of untried or rarely used features in CS-4 makes it difficult for a novice to learn and use the language and makes it difficult for those experienced

with other, smoother languages to become comfortable with CS-4. In many respects, it has the complexity of PL/I, but none of the defaults. It would appear that CS-4 would be among the most difficult of languages to learn (surpassing even ALGOL 68) and among the easiest to forget. The training investment for CS-4 would be considerable.

7. PROCEDURES AND DATA

The high order features of CS-4 makes a firm distinction between procedures and data. However, the ability to include machine code insertions eliminates the protection provided by this high level feature and permits the construction of self-modifying programs in the direct code sections. There is apparently no mechanism available which can prevent such an occurrence as long as one is permitted to use machine code; furthermore, there is no guarantee that these machine code insertions will limit their effects to a single module. The machine code could obtain the name of another procedure, load that name, hence obtaining its address, and make modifications within that procedure!

8. LANGUAGE SIZE

By most common measures, CS-4 is a very large language. It contains over 400 productions in the basic language syntax, excluding such things as I/O which are not part of the base language. There are more than 250 nonterminals in the syntax definition. There are nearly 300 keywords. The language is not merely large or baroque; it is best described as rococo! ("excessively ornate or intricate")

This size has serious implications not only for learning (as noted above) but for implementation. The language is presently being implemented for the first time on a large-scale computer. The feasibility of creating an efficient implementation of CS-4 on a small machine is highly questionable. There are some untested language features whose implementability

are not well known.

9. ANALYZABILITY

CS-4 programs lend themselves to analysis quite well except under error conditions, in which the control structure may become quite complex. There are only a couple of trouble spots, caused by the multi-level exits provided by the GO TO and ABORT TO statements. The RANGE feature of the language will automatically trap those variables which leave a certain predefined range. An interface to the operating system will permit various timing operations to be performed quite easily as well.

10. SUMMARY

The criteria used here are fundamental to language design and underlie much of the current research. While other criteria could easily be added, one can obtain a good feeling for the potential of the language by using this list alone.

One's initial impression of CS-4 is that it satisfies TINMAN quite well and that it tries to address many of the issues raised in TINMAN. The impression becomes increasingly worse as one studies the language, particularly in comparison with other research languages currently under development. The conclusion can only be that CS-4 makes some useful contributions to language design and achieves some language features which are missing from other languages, but the resulting whole leaves much to be desired.

## SECTION 2. EVALUATION OF CS-4 AGAINST THE TINMAN

## 1. OVERVIEW

CS-4 is being designed to conform to the requirements for programming languages being established by the DoD, most recently expressed in the TINMAN. Although it comes relatively close to the TINMAN on a strict evaluation of conformance to requirements, there are a number of problems with the language.

Its greatest strength from the standpoint of TINMAN is the support given to various built-in and extended data types, its exception handling mechanism, its ability to incorporate object representations for specific machine implementations, and its range/initialization facilities. Its greatest weaknesses are lack of recursion, lack of a pointer mechanism, absence of input/output and parallel processing from the base language, and lack of alternative structures in records.

There are many inconsistencies in the language, such as in the kind of types which may be used as parameters in different places. One is not given the impression of a language which was carefully designed as a synthesis of good language features. Instead, one receives the impression of starting with a basic set of language structures and adding features in ad hoc way as new requirements were added to the TINMAN, or as a clearer understanding of the TINMAN requirements was gained. The language lacks the smoothness and orthogonality of ALGOL 68 or PASCAL, which are built upon a more solid foundation involving a degree of formalism. CS-4 incorporates too many little-understood concepts.

Although there are some features of CS-4 that are handled particularly well and which could be incorporated into the language which is designed (or adapted) to meet the TINMAN (as revised), CS-4 does not seem to be a suitable base language for that effort. The language is already unwieldy, with far too many statements, keywords, and syntactic structures.

Adding the capability for recursion, pointers, and other "essential" language features would almost certainly lead to a language which could neither be used nor understood.

Part of the difficulty lies with the TINMAN rather than with CS-4 itself. A separate section "Notes on the TINMAN Requirements," points out some of the inconsistencies and problems with the TINMAN. It is to be expected that any language which attempted to meet all of the requirements would of necessity fail.

(It should also be observed that most well designed programming languages are the result of the work of one, or at most two, individuals. This is the case with ALGOL 68, PASCAL, LISP, APL, SNOBOL 4, etc., all of which, whatever their strengths or weaknesses, have made a contribution to programming languages. Those languages which have been designed by committee, with the notable exception of ALGOL 60, have the tendency to grow large and inconsistent without an adequate underpinning as the result of trying to meet the diverse views of their designers.)

## 2. DATA AND TYPES

CS-4 is a fully typed language. Built-in types include INTEGER, REAL, STATUS, FRACTION, STRING (FIXED and VARYING), SET, COMPLEX, and BOOLEAN. These may be formed into ARRAYs and STRUCTUREs. Union types may be formed. A one-dimensional array of reals may be declared as a VECTOR; a two-dimensional array of reals may be declared as a MATRIX. Both VECTOR and MATRIX types have a built-in set of operations associated with them.

There are no fixed point variables, violating TINMAN requirements A4 fully and A2 in part. Alternative structures in CS-4 are handled by declaring an object of UNION type within a structure along with the appropriate TAG field.

Declarations of objects are quite thorough, including a range part and an initialization part. Whereas one might declare

```
int a
```

in ALGOL 68, a comparable declaration in CS-4 is

```
VARIABLE A IS INTEGER (RANGE: MIN-PORTABLE-INTEGER-
VALUE THRU MAX-PORTABLE-INTEGER-VALUE) [NOVALUE];
```

Array declarations are handled similarly. Arrays may have any number of dimensions and may be of any mode, including ARRAY. A6 is not entirely met, though, since an array may be passed to a procedure with the upper and lower bounds designated by *, with the value supplied at block or procedure entry time.

3. OPERATIONS

CS-4 defines the meaning of assignment for every type, including a mechanism whereby assignment can be defined for user-defined types. However, the treatment of union types is not entirely consistent with this. In order to make an assignment within a union type, it is necessary to make certain that the tag of the union type corresponds to the types of the value which is to be assigned.

CS-4 considers the range specifications of an object as part of the type definition. In other words, two objects are of different types if their ranges are different. In that case, a run-time check must be made on the value to be assigned to make certain that it is within the range of the variable receiving the value. Otherwise an exception is signaled.

The "=" operator only applies for objects of the same type. One good feature of CS-4 with respect to relational operators is the capability to compare real numbers within a

precision. Since the declaration of reals includes a precision (possibly defaulted), the concept of equality, with respect to floating point numbers, takes on a different meaning. This distinction is quite significant in applications involving numerical computation where successive rounding errors may lead to an unexpected result, such as an extra iteration through a loop.

Scalar operations are not defined for real arrays (B7) and there is no way to extend those operators to achieve this objective. However, scalar operations are defined for objects of type VECTOR and MATRIX, i.e., for special one and two dimensional objects of type real. One may, in general, though make data transfers between structures or arrays with identical logical structures.

Another inconsistency in CS-4 shows up in the integer/character conversion. The ASCII function converts characters to integer but there is no inverse function for converting numerics to character.

No input/output operations (B10) are in the base language; all are in the operating system interface for the language.

4. EXPRESSIONS AND PARAMETERS

CS-4 does not specify the order in which expressions are evaluated other than to say that operators with the highest precedence are evaluated first. Since there is no restriction of side effects within functions, The ordering of expression evaluation could produce different results than would left-to-right. Since the language does not specify this ordering, various implementors can act accordingly, making whatever optimizations they see appropriate. This seems to be a rather serious omission from the language definition, especially since portability of programs is a goal both of CS-4 and TINMAN. (C1).

Another language inconsistency arises in conjunction with passing parameter. The parameters to a procedure are those which satisfy production 28 for primitive-mode-invocation, (p. 211) while the parameters for a mode declaration are those which satisfy production 206, restricted-mode-invocation (p. 223). It is not clear why the restriction was imposed for mode declarations.

Parameter passing itself is somewhat inconsistent as well. There are 10 classes of parameter passing, which may be grouped into 4 categories. There are three forms of parameter passing by COPY (similar to call by VALUE), three forms of parameter passing by REF (copy by REFERENCE), and three forms of parameter passing by NAME (copy by NAME). "Copy by Name" permits procedure and function names to be passed; it also permits expressions to be passes as in ALGOL 60 and thus leads to a mechanism for Jensen's device. Most of the binding attributes are straightforward except for COPYO, in which the actual parameter is <u>not</u> used to initialize the copy in the formal parameter; instead, such a parameter must be initialized. The 10th class of parameter passing is the attribute HANDLES, which accepts 1 or more exception names. C7 is thus not completely satisfied.

CS-4 requires the full description of all formal parameters. There is no default situation as suggested by C8 which would permit generic procedures; in fact, generic procedures are hard to construct in CS-4. It is not immediately clear as to the restrictions on union types being passed as parameters. A formal parameter may be of union type, but may not have a binding attribute of REFI, REFO, or REFIO. Apparently, though, it can be passed by NAME! It is not clear as to whether an object of UNION type may be passed as an actual parameter to a formal parameter having one of the constituent types or whether an object of simple type, being one of the constituents of a union type, can be passed as an actual parameter to a formal parameter of a UNION type of which the simple type is one of the constituent types.

This must be cleared up in the language semantics.

CS-4 has no mechanism for variable arguments, although the input/output procedures accept them. One cannot in general define procedures with such a facility. A restricted means for doing this would be with union types. No other techniques are immediately apparent since there is no pointer mechanism; one could create an artificial pointer mechanism using integer variables (or the equivalent) and then pass the integer indicating the beginning address of a linked list, but this is a rather clumsy solution.

5. VARIABLES, LITERALS, AND CONSTANTS

CS-4 meets these requirements quite well, save for the absence of fixed point variables (D4) and pointer mechanisms (D6). CS-4 provides a full range of initialization and range capabilities, including an initialization procedure INIT which will be defined by the user for extended types. The range and initialization parts are required for all variable declarations. The major objection is that they can be quite cumbersome to use in simple programs.

6. DEFINITION FACILITIES

CS-4 programmers have access to strong definitional facilities, including those of abstract data types. The user may define structures and unions of existing types, may incorporate those structures or unions in more complex types, including other structures and unions, and may associate a set of procedures and/or functions with the representation of an object so that the representation of the object is hidden.

Mode declarations are actually type generators, since a type can be passed as a parameter to a mode declaration. One could declare the mode stack and then declare objects

which are integer stacks, real stacks, or even stacks of stacks. (There does not appear to be a restriction on recursive mode definitions, even though there is a restriction on recursive procedure definitions!) These mode declarations include built in operation names ASSIGN, INIT, COMPARE, and TERM, which provide facilities for assignment, initialization, comparison, and finalization of objects of the defined type. The user may define these as desired. The set of names which is visible from outside the mode declaration is given in the CAPABILITY statement. (CAPABILITY is a rather unusual choice of word, since it has a different explicit and well-known meaning with respect to operating systems; EXPORTS seems like a more appropriate word to denote the names of operators which are exported from the mode declaration and made known outside the declaration.)

Unfortunately, the existing dyadic operators cannot be extended to these new types, nor is there a method for defining new infix operators. All use of user-defined types is through function and procedure calls, thereby violating E2.

CS-4 appears to be headed in the right direction here, but Euclid appears to have solved this set of problems more effectively, although it, too, fails on the required operator extension feature.

## 7. SCOPES AND LIBRARIES

CS-4 facilities with respect to libraries and compools is undetermined. These features are dependent upon the environment in which the program executes. The mechanism for external procedures seems well-suited for libraries, though.

Variables may be declared STATIC or AUTOMATIC or SHARED (with or without protection), with AUTOMATIC as the default case. Scope is determined at compile time for

STATIC and AUTOMATIC variables. The scope of a SHARED variable is determined within a procedure, but its scope includes other programs as well.

The mode declaration mechanism provides a set of capabilities (i.e., names) which can be used to access an object of a user-defined mode. F1 and F2 are met.

F7 is not met since there is no input/output in the base language.

8. CONTROL STRUCTURES

CS-4 has all of the basic control structures, but has no facilities for recursion, parallel processing, or asynchronous interrupt handling. However, parallel processing and interrupt handling can be accomplished through the operating system interface of the language.

CS-4 meets the requirements G2 and G3 partially. There are inadequate restrictions on the go to, since one can branch into an enclosing block as well as within the most local scope. Furthermore the EXIT statement, which is used to terminate loops, may branch through more than one nested loop construct, thereby terminating more than one loop. G3 is almost met as well, but the IF-THEN-ELSE is not fully partitioned, since there is no requirement that the ELSE be included. No ambiguity results, since the IF statement is terminated with a FI, whether or not the ELSE appears. The statement IF A THEN B FI may be seen as being equivalent to the fully partitioned statement IF A THEN B ELSE ; FI where the ";" represents a null statement.

The CS-4 exception-handling mechanism is reasonably powerful and well-controlled. Procedures are associated with exceptions and are named as handlers. The appropriate handler for a given exception is determined by dynamic scoping rules. Approximately 60 signals are built into CS-4 to which the

user may append an arbitrary number of user-defined signals. G7 is well met and the flow of control upon exceptions is well-restricted. CS-4 meets requirement G7 as well as any existing language, and significantly better than PL/1 does.

## 9. SYNTAX AND COMMENT CONVENTIONS

CS-4 is not a simple language. It thus fails to fully satisfy requirements H1 and H6. The grammar of the base language contains over 400 productions, with approximately 250 non-terminal symbols! A significant number of these productions exist as a result of inconsistencies in the language among the syntax forms. While the language can be parsed in a straightforward way, the tree is quite large and the language could not easily be implemented on small-to-medium sized computers. The language defines more than 250 keywords, which fall into three categories. The first category is 48 words with fixed meanings. Others are either words which have meaning to CS-4 in limited places only or which can be freely overridden by users. This is in contrast to most other languages, with the notable exception of PL/I.

H8, the restriction on parentheses, is met, except that a programmer might use a character string such as "ABC(", which contains an unmatched parenthesis. The syntax, though, does not permit unmatched parentheses.

As with most other languages, CS-4 fails to achieve a uniform reference mechanism. The structure addressing syntax uses a "." notation, while function calls, mode definitions, and array subscripts use parentheses. However, array elements could be replaced by function calls.

While there are few unique notations for special cases, the syntax of the language occasionally becomes quite complex, particularly when one is defining arrays or structures as parameters within procedures.

## 10. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

The CS-4 language meets this set of requirements quite well. There are facilities for providing machine structure definitions when the programmer wishes to override the default definition of a STRUCTURE type. Characteristics of the host machine may be associated with the program at compile time so that storage alignment and size can be properly worked out. This feature applies to STRUCTURE variables and not to other program types. However, one could declare a single variable within an MSTRUCTURE to achieve the desired effect.

The language base houses most of the power of the language. However, input/output and parallelism are in the operating system interface rather than in the language itself. The language base is far from being minimal, since there are often many alternative ways of doing the same thing; VECTOR and ARRAY of REAL with one dimension is one example. The size of the language and the rather poor presentation of the language in the reference document detract significantly form the understandability of the language. The language semantics are relatively poorly defined.

## 11. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

It is difficult to determine the potential efficiency of the language translator. The size of the grammar would appear to impose a significant compilation cost. Checking ranges and checking parameters of union type might impose run time costs not generally incurred with other languages. For example, one could write

```
VARIABLE I IS INTEGER (RANGE: 1 THRU 10) [1],
          J IS INTEGER (RANGE: 1 THRU 10) [1];
```

where the compiler might not see that there is no need for a run time check on an assignment of I to J or vice versa. One only hopes that the compiler would not build in run time checks on:

VARIABLE (I,J) IS INTEGER (RANGE: 1 THRU 10) [1];

It is quite likely though that the first instance would be treated as two different types simply because of the disjoint declarations.

The effect of optimizations (J2) is unknown in the absence of information about the ordering of expression evaluation and the prevention of side effects (C1).

CS-4 meets the other TINMAN requirements in this category. MPROCEDURES, MSTRUCTURES, and the OPEN and MOPEN features give the desired abilities. OPEN and MOPEN provide a generalized MACRO facility.

12. PROGRAM ENVIRONMENT

CS-4 requires an operating system interface since there is no input/output in the base language (K1). An operating system interface is necessary for integration of separate modules into the program; a requirement staisfied by the EPROCEDURE concept. EPROCEDURES permit not only CS-4 programs, but programs written in other languages, to be integrated with CS-4 programs. CS-4 provides a built-in capability for PL/I and FORTRAN as other HOL's, and a capability to specify routine linkage mechanisms for other calling protocols.

CS-4 is intended to run in an environment which provides useful tools for program creation and debugging. It should be noted, though, that the tools on the IBM 370, which is the initial target machine, leave a great deal to be desired.

Assertions may only be included as comments. CS-4 is not intended for verification. Indeed, there are no proof rules for CS-4 itself and no formal specification of the language semantics. There undoubtedly will be many surprises for CS-4 programmers as they determine how the language processor handles some of the language issues which are not clearly specified in the reference document.

## 13. TRANSLATORS

Very little can be said about CS-4 translators. The language is sufficiently large that one is tempted to eliminate certain features in a first implementation. Furthermore, minimization of compile time as an objective is difficult to achieve, for reasons which have been taken up in the section "Notes on the TINMAN Requirements."

It is expected that a CS-4 translator can be written for a number of different host computers, but there is some question as to whether such a translator can run on a relatively small computer. Some form of cross-compiler, as anticipated by L5, might be needed.

It is impossible to address L6 clearly, although it is apparent that the language provides the necessary features for full type and range checking.

It would be feasible to use CS-4 as an implementation language for a CS-4 translator (L9).

## 14. LANGUAGE DEFINITION, STANDARDS, AND CONTROL

TINMAN criterion M1 is not well met. Many of the features of CS-4 are not well understood; abstract data types, for example, have not been fully worked out. Many of the language notions, including FRACTION, RESIGNAL, MOPEN, and NORECALL, are only lightly used or are untried elsewhere. The impression is that these features were often added as patches to a previously designed language.

Criterion M2 is not well met. There is no unambiguous definition of semantics; much is left to interpretation, requiring the reader to be familiar with a large number of programming language concepts. Some semantic issues are not addressed; e.g., expression evaluation and parameter passing involving union types are incompletely explained.

Criterion M3 is partially met. There is a CS-4 primer, but there is not a "formal in-depth description." The CS-4 Language Reference Manual does not meet that requirement.

How well CS-4 meets the requirements of DoD control over translators, standards, and libraries is impossible to state. The developer of the language or the contracting agency could impose a sufficient set of controls to achieve the objective, particularly since the present implementation level of the language is still very low.

# SECTION 3. LANGUAGE EVALUATION SUMMARY FOR CS-4

HOL EVALUATION PROJECT:
LANGUAGE EVALUATION SUMMARY FOR CS-4

A. DATA AND TYPES

A01 S
A02 P
A03 S
A04 N
A05 P
A06 P
A07 S

B. OPERATIONS

B01 P
B02 N
B03 S
B04 P
B05 S
B06 S
B07 P
B08 P
B09 P
B10 N
B11 S

C. EXPRESSIONS AND PARAMETERS

C01 N
C02 S
C03 S
C04 P
C05 N
C06 S
C07 N
C08 N
C09 N

D. VARIABLES, LITERALS, AND CONSTANTS

D01 S
D02 S
D03 S
D04 P
D05 S
D06 N

E. DEFINITION FACILITIES

E01 S
E02 N
E03 S
E04 N
E05 P
E06 S
E07 S
E08 S

F. SCOPES AND LIBRARIES

F01 S
F02 S
F03 S
F04 U
F05 U
F06 U
F07 N

G. CONTROL STRUCTURES

G01 P
G02 P
G03 P
G04 S
G05 N
G06 N
G07 S
G08 N

H. SYNTAX AND COMMENT CONVENTIONS

H01 P
H02 P
H03 S
H04 S
H05 S
H06 N
H07 N
H08 S
H09 N
H10 S

I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

I01 S
I02 P
I03 S
I04 N
I05 P
I06 S
I07 S

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCE

J01 U
J02 U
J03 S
J04 S
J05 S

K. PROGRAM ENVIRONMENT

K01 N
K02 S
K03 U
K04 U
K05 P

L. TRANSLATORS

L01 U
L02 U
L03 U
L04 U
L05 U
L06 S
L07 S
L08 S
L09 S

M. LANGUAGE DEFINITION, STANDARDS, AND CONTROL

M01 N
M02 N
M03 S
M04 U
M05 U
M06 U

Key: S = Satisfied
P = Partially Satisfied
N = Not Satisfied
U = Undetermined

# SECTION 4. DETAILED COMPARISON OF CS-4 TO TINMAN

## 1. INTRODUCTION

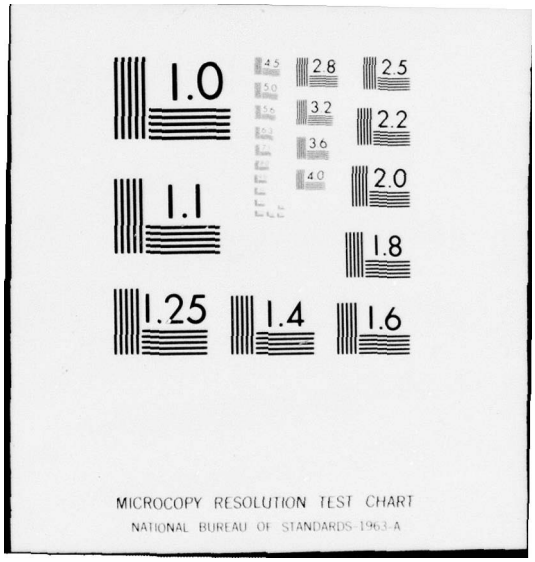
This section outlines specific evaluations of each TINMAN requirement as they are (or are not) met by the language CS-4.

Page number references where given refer to the basic reference document for this language: CS-4: Language Reference Manual and Operating System Interface, Intermetrics, Inc., IR-130-2, October 1975.

AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

5 OF 28
ADA037634


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

A. DATA AND TYPES

A1. Typed Language
Satisfied

A2. Built-in Data Types
Partially Satisfied

No fixed point

A3. Precision Declaration
Satisfied

A4. Fixed Point
Not Satisfied

No fixed point

A5. Enumeration Sense of Character Sets
Partially Satisfied

STATUS types handle this. EBCDIC not specified.

A6. Arrays
Partially Satisfied

Both lower and upper array bounds may be undetermined at compile time.

A7. Record Equivalence
Satisfied

Alternative structures may be created by declaring UNION types in structures.

B. OPERATIONS

B1. Assignment
Partially Satisfied

Assignment to an object of union type can only be from another object of union type. There are also restrictions concerning assignments when the ranges of two objects do not intersect.

B2. Equivalence Operator
Not Satisfied

The objects compared must agree in type.

B3. Relationals Defined
Satisfied

There are also functions for relations on reals to perform comparisons within a precision.

B4. Arithmetic Operations
Partially Satisfied

No fixed point

B5. Truncation/Rounding
Satisfied

B6. Boolean Operations
Satisfied

B7. Scalar Operations
Partially Satisfied

No scalar operations on conformable arrays. Scalar operations on REAL VECTORS. Data transfers between records or arrays of identical logical structure are permitted.

B8. Type Conversion
Partially Satisfied

No fixed point. ASCII function converts characters to integers, but there is no inverse. An actual parameter must be the same type as a formal parameter. If the formal parameter is of union type, the actual parameter (p. 146) must be of the same union type, not a constituent type.

B9. Range Exceptions
Partially Satisfied

No fixed point. X-INTEGER-RANGE may be signaled.

B10. I/O Operations
Not Satisfied

The base language has no input/output operations.

B11. Power Set Operations
Satisfied

C. EXPRESSIONS AND PARAMETERS

C1. Side Effects
Not Satisfied

C2. Operand Hierarchy
Satisfied

3 levels of relational, 4 levels of arithmetic hierarchy

C3. Expressions Permitted
Satisfied

C4. Constant Expressions
Partially Satisfied

Evaluation of constant expressions cannot be determined.

C5. Parameter Rules
Not Satisfied

Parameters exist for exception-handling, procedures, and declarations. The types of parameters that can be passed are different for each of the classes. Also, different kinds of binding are associated, with only the NAMEI binding class being meaningful for mode-invocations (pp211, 222).

C6. Parameter Agreement
Satisfied

C7. Formal Parameter Kinds
Not Satisfied

There are 9 binding attributes: COPYI, COPYO, COPYIO, NAMEI, NAMEO, NAMEIO, REFI, REFO, and REFIO. The HANDLES attribute passes exception names. This could be satisfied if, for example, COPYI, COPYO, and COPYIO were considered to form a class. However, objects other than procedure names may be passed by NAME.

C8. Formal Parameter Specifications Optional
Not Satisfied

C9. Variable Numbers of Parameters
Not Satisfied

D. VARIABLES, LITERALS, AND CONSTANTS

D1. Constant Value Identifiers
Satisfied

D2. Numeric Literals
Satisfied

D3. Initialization
Satisfied

D4. Ranges and Stepsize
Partially Satisfied

No fixed point variables

D5. Variable Types
Satisfied

D6. Pointer Variables
Not Satisfied

No pointer mechanism

E. DEFINITION FACILITIES

E1. Type, Operator Definitions
Satisfied

E2. Consistent Use
Not Satisfied

There is no facility for extending operators to new types.

E3. No Default Declarations
Satisfied

E4. Extend Existing Operators
Not Satisfied

E5. Type Definitions
Partially Satisfied

There is some inheritance of data operations.

E6. Data Definitions
Satisfied

E7. *No Free Union*
Satisfied

E8. Type Initialization and Allocation
Satisfied

F. SCOPES AND LIBRARIES

F1. Allocation and Access
Satisfied

Can have STATIC allocation

F2. Limit Access Scope
Satisfied

The operations associated with a separately defined structure are specified in a capability list. Only those names are available outside the mode declaration.

F3. Static Scope Determination
Satisfied

F4. Libraries Available
Undetermined

F5. Library Content
Undetermined

F6. Libraries and Compools
Undetermined

F7. Machine Dependent Interfaces
Not Satisfied

G. CONTROL STRUCTURES

G1. Control Structures
Partially Satisfied

No recursion, parallel processing, or interrupt handling in base language

G2. GOTO
Partially Satisfied

There is a goto, but the branching scope includes enclosing blocks as well as the most local scope.

G3. Conditional Control Structures
Partially Satisfied

The else is not required following an if then.

G4. Iteration Control
Satisfied

G5. Recursive Routines
Not Satisfied

No recursion (p. 140)

G6. Parallel Processing
Not Satisfied

G7. Exception Handling
Satisfied

(See p. 154)

G8. Synchronization Primitives
Not Satisfied

## H. SYNTAX AND COMMENT CONVENTIONS

H1. Syntax Characteristics
Partially Satisfied

The grammar is context free, but quite large, with over 400 productions involving approximately 250 non-terminals.

H2. No Syntax Extensions
Partially Satisfied

Some of the key words may be redefined, as identifiers or procedure names for example.

H3. Source Character Set
Satisfied

H4. Literals and Identifiers
Satisfied

For identifiers (__) and numeric literals (^) (p. 9)

H5. Continued Lexicals
Satisfied

(p. 5)

H6 Keywords
Not Satisfied

CS-4 has approximately 260 Keywords, divided into three categories. Category 1 are fixed. Category 2 have fixed meaning in certain contexts, but may be freely used elsewhere. Category 3 may be overridden by programs. There are 48 Category 1 Keywords (pp 261-3).

H7. Comment Convention
Not Satisfied

There are two comment conventions: ( ), which may be nested and may cover multiple lines, and % which treats the remainder of a line as comment.

H8. Unmatched Parentheses
Satisfied

The only exception is perhaps within string literals.

H9. Uniform Referent
Not Satisfied

The structures use a . notation, which is different from ( ) used by procedures and arrays.

H10. Consistency of Meaning
Satisfied

I. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

I1. Defaults in Program Logic
Satisfied

I2. Default Override
Partially Satisfied

The programmer may declare MSTRUCTURES for defining object representations at the machine representation level.

I3. Compile-time Parameters
Satisfied

I4. Object Environment Compile-time Parameter
Not Satisfied

No conditional compilations in source language

I5. Language Kernel
Partially Satisfied

VECTOR largely duplicates REAL ARRAY, and MATRIX largely duplicates REAL ARRAY with 2 dimensions. The base language, given its size, is not "simple."

I6. Translator Restrictions Defined
Satisfied

I7. Object Environment Restrictions
Satisfied

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCE

J1. Efficient Code
Undetermined

Depends on parsing techniques

J2. Optimization Invisible
Undetermined

J3. Access to Machine Language
Satisfied

MPROCEDURES (pp.194-5)

J4. Implementation Representation Tradeoffs
Satisfied

MSTRUCTURES

J5. Open/Closed Implementations
Satisfied

OPEN attribute for procedures

K. PROGRAM ENVIRONMENT

K1. Operating System Not Required
Not Satisfied

Language assumes OS interface for I/O, etc.

K2. Program Assembly
Satisfied

EPROCEDURES

K3. Software Development Tools
Undetermined

K4. Translator Options
Undetermined

K5. Assertion Capability
Partially Satisfied

Included only as comments

L. TRANSLATORS

L1. Language Supersets
Undetermined

L2. Language Subsets
Undetermined

L3. Low-Cost Translation
Undetermined

L4. Many Object Machines
Undetermined

L5. Self-Hosting
Undetermined

L6. Translator Checking
Satisfied

L7. Diagnostic Messages
Satisfied

CTWARNINGS option

L8. Translator Structure
Satisfied

L9. Self-Implementable
Satisfied

It would be possible. However, the absence of recursion and pointers detracts from the usability of some of the favored parsing methods.

M. LANGUAGE DEFINITION, STANDARDS AND CONTROL

M1. Existing Features
Not Satisfied

Many language notions are lightly used or untried elsewhere. CAPABILITY, FRACTION, RESIGNAL, MOPEN, NORECALL are just a few of these.

M2. Unambiguous Definition
Not Satisfied

The present language definition is not entirely clear on semantics. For example, order of expression evaluation is not specified.

M3. Language Documentation
Satisfied

M4. Control Agent
Undetermined

M5. Support Agent
Undetermined

M6. Library Standards
Undetermined

# CHAPTER 8
# REFERENCES

1. W.A. Whitaker, "Department of Defense Requirements for High Order Computer Programming Languages, Tinman", 1 March 1976.

2. P. Naur (Editor), "The Revised Report on the Algorithmic Language ALGOL-60". Communications ACM, January 1963.

3. J.B. Goodenough and L.H. Shafer, "A Study of High Level Language Features". SofTech/ECOM, ECOM-75-0373-F, Volumes 1 and 2.

4. J.D. Ichbiah, J.P. Rissen, and J.C. Heliard, "The Two-Level Approach to Data Independent Programming in the LIS System Implementation Language". Compagnie Inter. Pour L'Informatique, Louveciennes, France, June 1975.

5. The System Implementation Language LIS, CII. Document No. 4549 EL/EN, January 1976.

6. CS-4 Language Reference Manual and Operating System Interface. Advanced Software Technology Division, NELC, San Diego, California, October 1975.

7. CS-4 Primer, Volume I, Basic Features. Advanced Software Technology Division, NELC, San Diego, California, February 1975.

8. A.I. Wasserman, "Programming Language Designs and Programming Discipline". UCSF, Medical Information Science, San Francisco, November 1976.

9. A.I. Wasserman, "PLAIN: Reliable Interactive Software and Programming Language Design." Technical Report No. 20, UCSF, Laboratory of Medical Information Science, January 1977.

10. A. Van Wijngaarden, et. al., "Revised Report on the Algorithmic Language ALGOL 68". Acta Informatica, Volume 5, pages 1-236, 1975.

11. A.S. Tanenbaum, "A Tutorial on ALGOL 68". Computing Surveys, Volume 8, No. 2, June 1976.

12. R. Baumann, M. Feliciano, F.L. Bauer, and K. Samuelson, Introduction to ALGOL. Prentice-Hall, Englewood Cliffs, New Jersey, 1964.

13. J.D. Ichbiah, J.C. Heliard, "PLexes in LIS," 1975.

14. J.J. Ichbiah and P. Cousot, "Visibility and Separate Compilations," 1975.

15. Evaluation of ALGOL-68, JOVIAL J3B, PASCAL, SIMULA 67, and TACPOL vs. Tinman Requirements for a Common High Order Programming Language, Softech, Inc. Report No. 1021-14, October 1976.

16. R. Roessler and K. Schenk (Editors), "International Purdue Workshop on Industrial Computer Systems: A Language Comparison," Purdue Laboratory for Applied Industrial Control, Purdue University, October 1976.

17. R.M. De Morgan, I.D. Hill and B.A. Wichmann, "A Supplement to the ALGOL 60 Revised Report," The Computer Journal, Volume 19, No. 3, 1976.

18. C.M. Geschke and J.G. Mitchell, "On the Problem of Uniform References to Data Structures," 1975 International Conference on Reliable Software, pages 31 ff, April 1975.

19. Hoare, C.A.R., "Hints on Programming Language Design," Stanford University Computer Science Department CS-73-403, December, 1973.

SAI-78-523-110-SF

# HIGH ORDER LANGUAGE EVALUATION PROJECT

Final Report

Contract No. F030602-76-C-0165

Prepared For:

Rome Air Development Center
Griffiss Air Force Base
Rome, New York 13441

By:

E. F. Miller, Jr.
A. I. Wasserman

February 28, 1977

SCIENCE APPLICATIONS, INCORPORATED

Software Technology Center
211 Sutter St., San Francisco, CA 94108
(415) 421-9111

# PREFACE

This Final Technical Report contains comments, evaluations, and recommendations made by the HOL Evaluation Project Team at Science Applications, Inc. (SAI), Software Technology Center (STC), from November 1976 through February 1977.

Four languages were evaluated: ALGOL 60, ALGOL 68, LIS, and CS-4. Evaluations against TINMAN requirements on a line-by-line basis, against TINMAN desires, against language design criteria, and against baseline language selection criteria were completed.

In addition, this report comments upon the TINMAN requirements and makes recommendations concerning the direction of future efforts by the HOLWG and the methodology to be used in a language design and evaluation effort. This material is predicated on the assumption that the present work will lead to such an effort. A portion of this material was presented orally to Dr. Sera Amoroso in a technical meeting held at STC in San Francisco, California, 16 December 1976.

TABLE OF CONTENTS


| | | Page |
|---|---|---|
| CHAPTER 1. | OVERVIEW AND CONCLUSIONS | 1-1 |
| Section 1. | Overview and Summary of HOL Evaluation Project Results | 1-2 |
| | 1. Introduction | 1-3 |
| | 2. HOL Evaluation Project Description | 1-3 |
| | 3. Detailed Language Analysis | 1-4 |
| | 4. Analysis by Major TINMAN Areas | 1-6 |
| | 5. General Language Design Criteria | 1-8 |
| | 6. Toward Design of a New Language | 1-9 |
| CHAPTER 2. | METHODOLOGY | 2-1 |
| Section 1. | Methodology for Evaluation of High Order Languages | 2-2 |
| | 1. Introduction | 2-3 |
| | 2. Detailed Analysis of Language Requirements | 2-3 |
| | 3. Evaluation Against TINMAN's Major Categories | 2-4 |
| | 4. Evaluation Against Language Design Criteria | 2-4 |
| | 5. Base Language Criteria | 2-5 |
| Section 2. | Overall Language Evaluation Tabulations | 2-7 |
| | 1. Introduction | 2-8 |
| | 2. Language Summaries | 2-8 |
| Section 3. | Criteria for Evaluation of High-Order Languages (HOLs) | 2-12 |
| | 1. Introduction | 2-13 |
| | 1.1 Abstraction | 2-13 |
| | 1.2 Modularity | 2-14 |
| | 1.3 Linear Flow of Control | 2-14 |
| | 1.4 Control of Scope and Binding | 2-15 |
| | 1.5 Readability and Comprehensibility | 2-15 |
| | 1.6 Procedures and Data | 2-15 |
| | 1.7 Language Size | 2-15 |
| | 1.8 Analyzability | 2-16 |
| Section 4. | Criteria for TINMAN "Base Language" Selection | 2-17 |
| | 1. Introductory Notes | 2-18 |
| | 2. Language Design and Definition | 2-18 |
| | 3. Compiler Specification and Design | 2-18 |
| | 4. Compiler Implementation | 2-19 |
| | 5. Language Control and Maintenance | 2-19 |
| | 6. Commentary | 2-19 |

TABLE OF CONTENTS (Cont.)


| | | Page |
|---|---|---|
| CHAPTER 3. | TINMAN | 3-1 |
| Section 1. | Comments on the TINMAN Requirements | 3-2 |
| | 1. Introduction | 3-3 |
| | 2. On High Level Languages | 3-4 |
| | 3. Abstraction vs. Efficiency in TINMAN | 3-5 |
| | 3.1 Compile Time Costs | 3-6 |
| | 3.2 Optimization | 3-6 |
| | 3.3 Language Extensions | 3-7 |
| | 3.4 Expression Evaluation | 3-8 |
| | 3.5 Direct Code | 3-8 |
| | 4. Hiding of Types | 3-9 |
| | 5. Numeric Arithmetic | 3-10 |
| | 6. Data Structures | 3-11 |
| | 7. The GO TO | 3-11 |
| | 8. Extension of Operators | 3-12 |
| | 9. Programmability | 3-12 |
| | 10. Analyzability | 3-13 |
| | 11. Miscellaneous Issues | 3-14 |
| | 12. Summary | 3-15 |
| CHAPTER 4. | ALGOL 60 | 4-1 |
| Section 1. | Evaluation of ALGOL 60 Against Language Design Criteria | 4-2 |
| | 1. Introduction | 4-3 |
| | 2. Abstraction | 4-3 |
| | 3. Modularity | 4-3 |
| | 4. Linear Flow of Control | 4-4 |
| | 5. Control of Scope and Binding | 4-5 |
| | 6. Readability and Comprehensibility | 4-5 |
| | 7. Procedures and Data | 4-6 |
| | 8. Language Size | 4-6 |
| | 9. Analyzability | 4-6 |
| | 10. Summary | 4-7 |
| Section 2. | Evaluation of ALGOL 60 Against the TINMAN | 4-8 |
| | 1. Overview | 4-9 |
| | 2. Data and Types | 4-9 |
| | 3. Operations | 4-10 |
| | 4. Expressions and Parameters | 4-11 |
| | 5. Variables, Literals, and Constants | 4-12 |
| | 6. Definition Facilities | 4-12 |
| | 7. Scopes and Libraries | 4-12 |
| | 8. Control Structures | 4-13 |

TABLE OF CONTENTS (Cont.)


| | | Page |
|---|---|---|
| Section 2. | Evaluation of ALGOL 60 Against the TINMAN (Cont.) | |
| | 9. Syntax and Comment Conventions..... | 4-14 |
| | 10. Defaults, Conditional Compilation, and Language Restrictions..... | 4-15 |
| | 11. Efficient Object Representations and Machine Dependencies..... | 4-15 |
| | 12. Program Environment..... | 4-16 |
| | 13. Translators..... | 4-17 |
| | 14. Language Definition, Standards, and Control..... | 4-17 |
| Section 3. | Language Evaluation Summary for ALGOL 60..... | 4-19 |
| Section 4. | Detailed Comparison of ALGOL 60 to the TINMAN. | 4-21 |
| | 1. Introduction..... | 4-22 |
| CHAPTER 5. | ALGOL 68..... | 5-1 |
| Section 1. | Evaluation of ALGOL 68 Against Language Design Criteria..... | 5-2 |
| | 1. Introduction..... | 5-3 |
| | 2. Abstraction..... | 5-3 |
| | 3. Modularity..... | 5-4 |
| | 4. Linear Flow of Control..... | 5-4 |
| | 5. Control of Scope and Binding..... | 5-5 |
| | 6. Readability and Comprehensibility..... | 5-5 |
| | 7. Procedures and Data..... | 5-6 |
| | 8. Language Size..... | 5-6 |
| | 9. Analyzability..... | 5-7 |
| | 10. Summary..... | 5-7 |
| Section 2. | Evaluation of ALGOL 68 Against the TINMAN.... | 5-8 |
| | 1. Overview..... | 5-9 |
| | 2. Data and Types..... | 5-10 |
| | 3. Operations..... | 5-11 |
| | 4. Expressions and Parameters..... | 5-14 |
| | 5. Variables, Literals, and Constants..... | 5-17 |
| | 6. Definition Facilities..... | 5-18 |
| | 7. Scopes and Libraries..... | 5-19 |
| | 8. Control Structures..... | 5-20 |
| | 9. Syntax and Comment Conventions..... | 5-21 |
| | 10. Defaults, Conditional Compilation, and Language Restrictions..... | 5-23 |
| | 11. Efficient Object Representation and Machine Dependencies..... | 5-24 |

TABLE OF CONTENTS (Cont.)


| | | Page |
|---|---|---|
| Section 2. | Evaluation of ALGOL 68 Against the TINMAN (Cont.) | |
| | 12. Program Environment | 5-24 |
| | 13. Translators | 5-24 |
| | 14. Language Standards, Definition, and Control | 5-25 |
| Section 3. | Language Evaluation Summary for ALGOL 68 | 5-27 |
| Section 4. | Detailed Comparison of ALGOL 68 to the TINMAN | 5-29 |
| | 1. Introduction | 5-30 |
| CHAPTER 6. | LIS | 6-1 |
| Section 1. | Evaluation of LIS Against Language Design Criteria | 6-2 |
| | 1. General Comments | 6-3 |
| | 2. Abstraction | 6-4 |
| | 3. Modularity | 6-5 |
| | 4. Linear Flow of Control | 6-5 |
| | 5. Control fo Scope and Binding | 6-6 |
| | 6. Readability and Comprehensibility | 6-6 |
| | 7. Procedures and Data | 6-7 |
| | 8. Language Size | 6-7 |
| | 9. Analyzability | 6-7 |
| Section 2. | Evaluation of LIS Against the TINMAN | 6-9 |
| | 1. Overview | 6-10 |
| | 2. Data Types | 6-11 |
| | 3. Operations | 6-12 |
| | 4. Expressions and Parameters | 6-13 |
| | 5. Variables, Literals, and Constants | 6-13 |
| | 6. Definition Facilities | 6-14 |
| | 7. Scopes and Libraries | 6-15 |
| | 8. Control Structures | 6-15 |
| | 9. Syntax and Comment Conventions | 6-16 |
| | 10. Defaults, Conditional Compilation, and Restrictions | 6-18 |
| | 11. Efficient Object Representations and Machine Dependencies | 6-18 |
| | 12. Program Environment | 6-19 |
| | 13. Translators | 6-19 |
| | 14. Language Definition, Standards and Control | 6-20 |

TABLE OF CONTENTS (Cont.)


| | | Page |
|---|---|---|
| Section 3. | Language Evaluation Summary for LIS..... | 6-21 |
| Section 4. | Detailed Comparison of LIS to the TINMAN..... | 6-23 |
| | 1. Introduction..... | 6-24 |
| CHAPTER 7 | CS-4..... | 7-1 |
| Section 1. | Evaluation of CS-4 Against Language Design Criteria..... | 7-2 |
| | 1. Introduction..... | 7-3 |
| | 2. Abstraction..... | 7-3 |
| | 3. Modularity..... | 7-4 |
| | 4. Linear Flow of Control..... | 7-5 |
| | 5. Control of Scope and Binding..... | 7-6 |
| | 6. Readability and Comprehensibility..... | 7-7 |
| | 7. Procedures and Data..... | 7-8 |
| | 8. Language Size..... | 7-8 |
| | 9. Analyzability..... | 7-9 |
| | 10. Summary..... | 7-9 |
| Section 2. | Evaluation of CS-4 Against the TINMAN..... | 7-10 |
| | 1. Overview..... | 7-11 |
| | 2. Data and Types..... | 7-12 |
| | 3. Operations..... | 7-13 |
| | 4. Expressions and Parameters..... | 7-14 |
| | 5. Variables, Literals, and Constants..... | 7-16 |
| | 6. Definition Facilities..... | 7-16 |
| | 7. Scopes and Libraries..... | 7-17 |
| | 8. Control Structures..... | 7-18 |
| | 9. Syntax and Comment Conventions..... | 7-19 |
| | 10. Defaults, Conditional Compilation, and Language Restrictions..... | 7-20 |
| | 11. Efficient Object Representations and Machine Dependencies..... | 7-20 |
| | 12. Program Environment..... | 7-21 |
| | 13. Translators..... | 7-22 |
| | 14. Language Definition, Standards, and Control..... | 7-22 |
| Section 3. | Language Evaluation Summary for CS-4..... | 7-24 |
| Section 4. | Detailed Comparison of CS-4 to TINMAN..... | 7-26 |
| | 1. Introduction..... | 7-27 |
| CHAPTER 8. | REFERENCES..... | 8-1 |

# CHAPTER 1
# OVERVIEW AND CONCLUSIONS

The material in this chapter provides overview information about the methodology employed in this project and the conclusions drawn from the work.

# SECTION 1. OVERVIEW AND SUMMARY OF HOL EVALUATION PROJECT RESULTS

## 1. INTRODUCTION

This report evaluates how well ALGOL 60, ALGOL 68, LIS, and CS-4 satisfy specific and general TINMAN requirements, assesses their compliance with criteria of good language design, and analyzes their suitability as a base language for extension and/or modification to meet HOLWG needs.

## 2. HOL EVALUATION PROJECT DESCRIPTION

The SAI HOL Evaluation Project had four distinct phases. The first phase involved detailed analysis of the candidate languages against the specific requirements of the TINMAN to determine the extent each of the subject languages satisfied (or did not satisfy) the requirements. The analyses produced "scores" showing the degree to which each language satisfied, partially satisfied, or did not satisfy, TINMAN's specifics.

The second phase considered each language in the context of evaluation areas suggested by the major sections within TINMAN. The intent was to evaluate the conformity of each language with the spirit of the TINMAN requirements, in hope of finding a suitable base language for the TINMAN, or finding one that might be easily modified to fit that role. Certain languages addressed specific areas of concern quite well, despite having relatively low overall "scores." Also, certain languages follow the ideas expressed in (or implied by) the TINMAN, but possessed or lacked features which resulted in only partial satisfaction of a specific requirement.

The third phase involved drafting a set of general language design criteria which abstracted the intent of the TINMAN regarding a number of language features. The goal was to determine if there was a good base language candidate, and whether the individual languages had particular characteristics that should be included in an eventual DoD

language. A secondary activity during this phase was a detailed analysis of the TINMAN, which resulted in some suggestions for changes and clarifications.

The fourth phase dealt with the need to design a "new" programming language using an existing language as a base and simultaneously examined some language features that should be included in it.

One result of this work was the recognition that none of the subject languages, nor any other programming language with which the Project Team is familiar, fully meets a majority of the TINMAN requirements.

It was for this reason that the later phases were undertaken, which addressed the issues of modifying the TINMAN to specify a feasible set of language design objectives and requirements. An effort was made to identify languages which handled certain extended requirements particularly well. It is anticipated that suitable base languages will include ALGOL 68, PASCAL, and (possibly) PL/I, so that selected features can be integrated into the framework of those langauges.

## 3. DETAILED LANGUAGE ANALYSIS

The work in the first phase involved comparing the features of ALGOL 60, ALGOL 68, LIS, and CS-4 with the requirements of the TINMAN, using available documentation on the various languages. Each language was rated against each of the TINMAN requirements in an objective manner. Following the initial analyses, each evaluation was repeated to make certain that it was consistent. This step was necessary because some of the TINMAN's requirements are quite complex and involve a number of language features, and because understanding of certain TINMAN requirements only became clear on this second pass.

The results show that the languages satisfied the

TINMAN criteria in the following order (from best to worst): CS-4, ALGOL 68, LIS, ALGOL 60. However, CS-4, the "best" of the languages, only fully satisfied 42 of the 98 requirements. A second measure, determining the number of requirements that languages completely failed to satisfy (intended as a measure of the ease of modification) led to the following ordering: LIS, CS-4, ALGOL 68, ALGOL 60. CS-4 again ranked high, failing to satisfy 22 of the requirements; even so, CS-4 was the weakest of the languages considered for several of the TINMAN requirement areas.

It is necessary to caution the reader against making a literal interpretation of these scores. Other organizations evaluating the same languages in the same manner would likely produce somewhat different results. There are several reasons for this. First, not all of the language definitions are entirely precise; for example, both the CS-4 and the LIS documents are "reference manuals" rather than more formal language definitions. CS-4, in particular, is still being designed and consequently leaves many questions unanswered. Second, there is a degree of subjectivity in distinguishing between completely satisfying and partially satisfying a TINMAN requirement. If a language conforms to the idea behind the requirement and fails to meet one minor point, it is possible to justify either an "S" or a "P" evaluation. This is especially true if a feature is missing, but the requirement is very easy to achieve with other existing language features.

Third, there is some room for interpretation of meaning within the TINMAN requirement. Different individuals have different ideas as to what constitutes "very few" keywords or a "minimal" base language, or whether safe and effective optimization is possible. In summary, there is a danger that subjective issues could be used to make one language look particularly viable as a candidate while downgrading other languages.

The self review of the detailed evaluations was intended to guarantee internal consistency among them.

## 4. ANALYSIS BY MAJOR TINMAN AREAS

It became quickly apparent from study of the TINMAN that no language exists which satisfied all of the requirements, or even a majority of them. The analysis was extended informally to some other languages, including PASCAL, Euclid, and some new languages still under design. This task led to confirmation of this conclusion. Accordingly, a search was begun to determine the suitability of languages to serve as a base of modification to ultimately meet the TINMAN requirements.

In this phase, the TINMAN requirements were grouped by areas (Data Types, etc.) and, at a somewhat less detailed level, the conformity between a given language and the key concepts contained in that area of the TINMAN were considered. Thus, if a language might have received a low score for failing to provide fixed point numbers, it might still rate high for being strongly typed, for providing a variety of data types, and for supporting compile-time type checking.

At this stage, the evaluation became less quantifiable. The result is a set of impressions of the extent to which certain ideas in TINMAN are captured in a given language. To some degree, one obtains a better "feeling" for the utility of a programming language at this level than at the more detailed level. It is necessary to examine sample programs to see how one might construct certain statements in the language. Knowledge of other programming languages and of programming methodology also comes into play.

This phase was extremely important, since the numerical scoring from the detailed analysis phase was indeed somewhat arbitrary and not especially useful once it had been concluded that no languages really came close to what was desired.

At this stage, one sees that ALGOL 68, despite the complexity of the language definition and a few complications related to elaboration of expressions, has great power and is highly consistent. The orthogonality of the various features of ALGOL 68 makes it possible to add new (orthogonal) features, or to delete unnecessary or unwanted ones, without major impact.

Similarly, LIS offers two elegant concepts: the PLEX notion, and the "data independent" type definition. The LIS approach to lower level features is also quite appealing, being handled more cleanly than in other languages.

The gross weaknesses of ALGOL 60 are also clear at this stage, thereby relegating ALGOL 60 to a place of historical interest (with significant impact on PASCAL and ALGOL 68) but eliminating it as a serious contender.

Finally, one sees that CS-4 has serious flaws and inconsistencies caused by the absence of a strong theoretical foundation for the language. Although this judgment may seem strong, CS-4 suggests itself as a set of language features added onto one another as more language design requirements were imposed, rather than having derived from a careful study and synthesis of fundamental programming language concepts. These weaknesses seem to make CS-4 somewhat less suitable as a basis for modifications than either PASCAL or ALGOL 68. CS-4 seems comparable to PL/1 in this sense, but lacks the experience gained from programming and implementation.

It was concluded that only ALGOL 68 (of the four languages being studied carefully) could serve as the base for language modification. However, the changes that would have to be made to ALGOL 68 are quite substantial, including major changes to incorporate abstract data types, enumeration types, exception and interrupt handling. A significant number of deletions

would have to be made as well. The conclusion is that adoption of a standard language by DoD would involve a substantial effort in language design, either by beginning from scratch or (more likely) making an extensive set of changes within the framework of an existing language. However, the resulting language might not resemble the base language very closely upon completion of that effort.

## 5. GENERAL LANGUAGE DESIGN CRITERIA

The third phase of the study attempted to use the TINMAN criteria as a basis for direct language design. It was felt that the list of TINMAN requirements was directed toward specific language features rather than toward broader language goals. Further, it was felt that goals for language design were important. The language design criteria established in this phase bore some resemblance to those presented in the introduction to the TINMAN, but had somewhat less emphasis on efficiency.

At the same time, a different view was taken of the TINMAN requirements proper. Whereas they had previously been treated as a given constant, they were now regarded as being subject to modification as well. (The recent release of IRONMAN substantiates this premise.)

A separate chapter was written which addressed some of the internal conflicts in TINMAN, with emphasis upon the difficulty of supporting high-level abstract notions on one hand and supporting low-level machine dependent notions on the other.

The four languages were then evaluated against the following set of language design criteria: abstraction, modularity, scope and binding of variables, linear flow of control, program readability, separation of procedures and data, language size, and program analyzability. These criteria were thought to

capture much of the essence of TINMAN, i.e., a language which satisfied the TINMAN requirements would rate relatively high when evaluated in this way.

This evaluation stage was instructive in that it showed some of the shortcomings of the TINMAN requirements, and identified some potential problems with any language which might be developed as the outcome of this effort. First, the resulting language is likely to be quite large by most measures. It will certainly be larger than PASCAL, FORTRAN, and ALGOL 60, simply because it tries to cover an extremely broad set of applications. It seems that the language may be comparable to PL/1 in size, although one would hope for more coherence and uniformity. However, there is a serious question as to the feasibility of implementing this language on any but the largest computers (recall that subsets are explicitly forbidden in TINMAN).

Second, the requirement for an interface to machine code, despite any protection that may be built in, seriously degrades the degree to which any language could satisfy the TINMAN criteria. The availability of "direct code" has a major impact on the areas of modularity, scope and binding of variables, procedures vs. data, linear flow of control, and program readability. In short, any program which makes use of machine code can subvert the intent of many important TINMAN requirements. If the final language contains the facility to incorporate machine code directly, the language will probably fail to satisfy a number of the other requirements of the language.

## 6. TOWARD DESIGN OF A NEW LANGUAGE

The last phase was undertaken because it was clear there was a need to synthesize a new programming language using features, as far as possible, from existing languages. One of the oft-repeated credos of programming language design

is to use language features which are known and understood, rather than attempting to design and incorporate a large set of new features into a language. [19]

In studying the four languages, it was apparent that some objectives are achieved very well in some of the languages, while other features are poorly handled in all of the languages. However, the scoring method used fails to show this distinction. For example the type extension and operator extension features of ALGOL 68 closely approximate the requirements of TINMAN, while the other languages do not even come close. Similarly, CS-4 and LIS attempt to address the lower level machine dependent interfaces, while ALGOL 60 and ALGOL 68 do not.

The last major activity was the identification of features in the languages evaluated which should be given strong consideration for synthesis into the eventual language design. This activity was predicated on the choice of an ALGOL-like base language (i.e., ALGOL 68, PASCAL, or PL/1) to be modified for the TINMAN requirements (or, more precisely, for a revised set of TINMAN requirements).

The effort did not attempt to create a new language, although the conclusion reached indicated this to be necessary. There is a very substantial difference between the identification of features which should be suitable for such a language, and the synthesis of those features into a coherent, formally defined language which fulfills the TINMAN requirements. The design of a new language is further complicated by the fact that a number of TINMAN requirements are not handled well in any existing programming language. Among these requirements are fixed point numbers and the avoidance of truncation, exception handling, asynchronous interrupt handling, and tradeoffs between compilation and runtime efficiency.

It is expected that extrapolation of these findings to the other languages under study by the HOLWG would confirm that no existing language meets the TINMAN requirements and that a language design effort must be undertaken. It is likely that a directed effort toward the design of a language which meets a revised set of TINMAN requirements would end up producing several candidate languages. None of these languages would fully meet all of the requirements, but all would be more satisfactory than any of the languages evaluated here. Such candidate languages would be able to draw not only on established languages, but also upon a number of research languages (CLU, ALPHARD, MESA, GYPSY, PLAIN, etc.) and proposed language features as well.

If such an effort were undertaken, the methodology developed here would be useful in analyzing newly designed candidate languages against a revised set of language requirements, eventually leading to a single language having DoD control and support.

# CHAPTER 2
# METHODOLOGY

The sections in this chapter relate to criteria that were used in the evaluations performed and to the overall results obtained in those evaluations.

Section 1. HOL Evaluation Project Methodology

Section 2. Overall Language Evaluation Tabulations

Section 3. Criteria for Evaluation of High-Order Languages (HOLs)

Section 4. Criteria for TINMAN Base Language Selection

# SECTION 1. METHODOLOGY FOR EVALUATION OF HIGH ORDER LANGUAGES

## 1. INTRODUCTION

This section describes a staged methodology that was used in the systematic evaluation of the properties of High Order Languages (HOLs) as they relate to the TINMAN. The methodological steps were, briefly:

- Identification of the role of the Tinman
- Item by item comparison of the Tinman to a specific language
- Evaluation of a language in terms of the Tinman's major categories
- Evaluation of a language against a specified set of language design criteria
- Evaluation of a language against a set of carefully designed base language selection criteria

The intended result of this methodology is the selection of one, or more, candidate languages to serve as a base language for further studies. Accomplishing this evidently implies simultaneously assessing potential modifications of the TINMAN requirements, and investigating the possibilities for extending candidate languages. Ultimately the modified TINMAN requirements and an extended candidate language will converge.

## 2. DETAILED ANALYSIS OF LANGUAGE REQUIREMENTS

Individual languages were compared on a TINMAN requirement-by-requirement basis. The basic question asked was: "Are the requirements of the TINMAN met?" The answers were organized into these classes:

- Satisfied - the language being analyzed was found to meet the TINMAN requirement completely.
- Partially Satisfied - at least one language property did not meet the TINMAN requirement, although other language properties did.
- Not Satisfied - the language did not satisfy the TINMAN requirement in any way.
- Undetermined (Undeterminable) - for various reasons the answer to the basic question could not be determined.

This latter category would apply for a number of reasons, mostly lack of documentary information or inapplicability of the requirement.

3. EVALUATION AGAINST TINMAN'S MAJOR CATEGORIES

This step in the methodology was concerned with a higher level assessment of the intentions of the TINMAN and the level to which particular languages met (or did not meet) those intentions.

Analyses at this level involved generalized discussions of language properties, including such issues as style of organization and/or structuring of language properties. Much of the basis for these analyses existed already in the discussions supporting each TINMAN requirement, as well as elsewhere in the TINMAN documentation.

4. EVALUATION AGAINST LANGUAGE DESIGN CRITERIA

The next stage in the sequence of analyses involved addressing the general goodness of a particular language in very general terms, not necessarily implying either match or mismatch with TINMAN objectives.

The criteria themselves were chosen as categories of investigation and critical comment that have a high impact on the ultimate utility of the language. The topics chosen were:

1. Abstraction--the capability of the language to cleanly represent separable "parts" of a solution to a problem.

2. Modularity--the contribution the language makes to the understandability and production ease (by humans) of programs written in it.

3. Linear Flow of Control--the contribution the language makes to ease with which the control is self-evident to a programmer.

4. Control of Scope and Binding--the facilities the language provides to minimize the complexity of programming in the presence of complex data structures.

5. Readability and Comprehensibility--the degree to which the language "reads" to a proficient programmer.

6. Procedures and Data--the measure of separation provided within the language between program and data structure.

7. Language Size--a measure of the difficulty a competent programmer has in learning the language from scratch.

8. Analyzability--the extent to which the language is amenable to automated program analysis tools used to assure software quality.

## 5. BASE LANGUAGE CRITERIA

The next stage in the methodology was to evaluate each candidate language against a set of criteria that could be used to determine its suitability for use as a base language against which additional language definition and refinement work could proceed. In accomplishing this it was recognized from the outset that, potentially at least, all of the major stages of language development must be represented.

The criteria selected reflected, therefore, the "normal" process of developing a language. The major elements that were involved were the following:

- Language Design and Definition. This category includes assessments of language style, language syntax, language semantics (including descriptive mechanisms), and tutorial aspects of language use.

- Compiler Specification and Design. This category includes assessments of the difficulty of building a compiler (translator) for the candidate language, estimates the level of effort needed to develop a detailed compiler specification, an assessment of algorithms needed within the compiler as they are implied by language features, and overall compiler design.

- Compiler Implementation. These assessments deal with the overall costs likely to be incurred in a compiler implementation (or compiler modification) activity. They deal also with the impact of requirements for self-boot, optimization, interactive operation, and program analysis capabilities.

- Language Control and Maintenance. This area can be partitioned into sub-areas that address existing dissemination of the language (assuming the language already exists), potential problems of controllability, and compiler validation techniques. Also included are assessments of control of subsidiary features.

# SECTION 2. OVERALL LANGUAGE EVALUATION TABULATIONS

## 1. INTRODUCTION

This section presents summary material about the results of the language evaluations that are reported in more detail elsewhere. Each TINMAN requirement was rated according to whether, for a particular language, it was satisfied (S), partially satisfied (P), not satisfied (N), or undetermined or undeterminable (U).

In some cases a requirement was classified as undetermined when there was insufficient information available to the evaluation team at the time of the evaluation; this should not necessarily imply that such information does not exist. The partially-satisfied category was assigned when at least one instance of a language property did not explicitly meet the TINMAN requirement or requirement part. It is important to note that in these cases, the judgments were deliberately harsh in terms of the "spirit" of the TINMAN.

## 2. LANGUAGE SUMMARIES

Tables 1 and 2 present the outcome of the evaluations, summarized by major TINMAN requirement section and by individual language. Table 1 indicates those instances where the TINMAN is explicitly satisfied; Table 2 indicates those instances where the TINMAN is explicitly not satisfied.

In rank ordering for those requirements satisfied, the four languages are, in decreasing order of precedence:

Requirements satisfied: CS-4, ALGOL 68, LIS, ALGOL 60

On the other hand, the rank ordering for these languages in terms of the requirements that are not satisfied, in increasing order of precedence, is:

Requirements not satisfied: LIS, CS-4, ALGOL 68, ALGOL 60.

It can be seen that based on this method of evaluation, CS-4 appears to rate "best" overall, since it scores highest for satisfying TINMAN requirements, while at the same time comes a close second best when the scores for not satisfying the TINMAN requirements are compared. On the other end of the scale, ALGOL 60 is farthest from satisfying, and closest to not satisfying the TINMAN requirements.

Table 1. HOL Evaluation Project
Overall S-Comparisons

| Topic | ALGOL 60 | ALGOL 68 | LIS | CS-4 | Weight |
|---|---|---|---|---|---|
| A. DATA AND TYPES | 1 | 1 | 2 | 3 | 16 |
| B. OPERATIONS | 2 | 5 | 5 | 4 | 8 |
| C. EXPRESSIONS AND PARAMETERS | 3 | 1 | 4 | 3 | 11 |
| D. VARIABLES, LITERALS, AND CONSTANTS | 0 | 4 | 5 | 4 | 7 |
| E. DEFINITION FACILITIES | 2 | 5 | 2 | 5 | 14 |
| F. SCOPES AND LIBRARIES | 2 | 3 | 2 | 3 | 8 |
| G. CONTROL STRUCTURES | 0 | 1 | 0 | 2 | 11 |
| H. SYNTAX AND COMMENT CONVENTIONS | 5 | 4 | 6 | 5 | 8 |
| I. DEFAULTS, CONDITIONAL COMPILATION, RESTRICTIONS | 3 | 2 | 3 | 4 | 8 |
| J. EFFICIENT OBJECT REPRESENTATION, MACHINE DEPENDENCE | 1 | 0 | 1 | 3 | 9 |
| K. PROGRAM ENVIRONMENT | 1 | 2 | 3 | 1 | - |
| L. TRANSLATORS | 0 | 2 | 3 | 4 | - |
| M. LANGUAGE DEFINITION, STANDARDS, CONTROL | 2 | 3 | 1 | 1 | - |
| TOTALS (Maximum = 98) | 22 | 34 | 37 | 42 | |

Table 2. HOL Evaluation Project
Overall N-Comparisons

| Topic | ALGOL 60 | ALGOL 68 | LIS | CS-4 | Weight |
|---|---|---|---|---|---|
| A. DATA AND TYPES | 5 | 1 | 4 | 1 | 16 |
| B. OPERATIONS | 5 | 2 | 4 | 2 | 8 |
| C. EXPRESSIONS AND PARAMETERS | 3 | 3 | 3 | 5 | 11 |
| D. VARIABLES, LITERALS, AND CONSTANTS | 4 | 1 | 1 | 1 | 7 |
| E. DEFINITION FACILITIES | 5 | 2 | 4 | 2 | 14 |
| F. SCOPES AND LIBRARIES | 2 | 1 | 0 | 1 | 8 |
| G. CONTROL STRUCTURES | 4 | 2 | 3 | 3 | 11 |
| H. SYNTAX AND COMMENT CONVENTIONS | 2 | 1 | 1 | 3 | 8 |
| I. DEFAULTS, CONDITIONAL COMPILATION, RESTRICTIONS | 3 | 2 | 1 | 1 | 8 |
| J. EFFICIENT OBJECT REPRESENTATION, MACHINE DEPENDENCE | 3 | 4 | 1 | 0 | 9 |
| K. PROGRAM ENVIRONMENT | 2 | 1 | 0 | 1 | - |
| L. TRANSLATORS | 2 | 3 | 0 | 0 | - |
| M. LANGUAGE DEFINITION, STANDARDS, CONTROL | 2 | 3 | 0 | 2 | - |
| TOTALS (Maximum = 98) | 42 | 26 | 20 | 22 | |

# SECTION 3. CRITERIA FOR EVALUATION OF HIGH-ORDER LANGUAGES (HOLs)

# 1. INTRODUCTION

Design goals for computer programming languages vary considerably, as a function of developer's tastes and the needs the language is intended to fulfill. Thus, it is unlikely that any particular set of evaluation criteria will be sufficient (or acceptable) when a number of languages will be compared against those criteria. Fortunately, however, there is some agreement on what are "good" features for an HOL to have; a set of eight such factors can form the basis for initial analyses of the suitability of a candidate language [8].

The eight factors are:

- Abstraction
- Modularity
- Linear Flow of Control
- Control of Scope and Binding
- Readability and Comprehensibility
- Procedures and Data
- Language Size
- Analyzability

The following sections give detailed explanations of the meaning of each of these factors.

## 1.1 Abstraction

Abstraction has been recognized as a means to develop a representation of concepts which relates closely to the application being programmed, to hide inessential details of the problem solution at various levels of the program development process, and to support the notion of "top-down" design. If a problem solution involves the use of stacks or trees, for example, one should be able to make use of those objects in the programming process. The ability to define these abstract objects, along with appropriate operations on the objects, is extremely valuable. If the objects and their associated operators are encapsulated so that the representation of the object is hidden from other parts of the program, the facility for data abstraction is comparable to the facility for procedural abstraction provided by

functions and procedures in many programming languages. Such a facility, termed abstract data types, provides the programmer with opportunity to refine program and data structures in parallel and makes it possible to create data objects within a program which resemble those used in the problem solution, thereby easing the process of transforming the problem solution into a program.

## 1.2 Modularity

Modularity has been observed to make an important contribution to the overall comprehensibility of programs, to the practice of programming by levels of abstraction, and to the production of large software systems by allowing various pieces of a software system to be effectively isolated from one another in a conceptually meaningful way. The ability to decompose a large problem into a number of smaller ones and to clearly delineate the interactions among the pieces in an important tool in gaining intellectual mastery over complex problems. Programming aids such as structure charts and HIPO charts have been developed to help the programmer identify modules and represent the total structure of the software system. A module can correspond to either a procedural abstraction or a data abstraction.

## 1.3 Linear Flow of Control

Linear flow of control within a program module has been recognized to be important for its value in debugging, verification and the understanding of program operation. Formal verification of programs and program readability are both simplified by controlling the branching within a program and by reducing the number of possible paths through a program. As a result, the use of unrestrained transfers of control (GOTOs) has fallen into disrepute and many alternative and more powerful control structures have been proposed.

## 1.4 Control of Scope and Binding

Control of scope and binding of variables has been identified as a technique which reduces programming errors caused by side effects and which serves as an aid to verification. Such control is necessary in order to achieve modularity on a formal sense. Without it, the programmer may easily circumvent restrictions concerning the proper use of input and output parameters for a module.

## 1.5 Readability and Comprehensibility

Readability and comprehensibility of programs has been seen to be a valuable program property which contributes to ease of program maintenance and modification. The use of opaque programming tricks or the construction of cryptic programs is no longer considered to be an acceptable programming practice. Many properties combine to yield readable programs, including the use of mnemonic variable names, the liberal insertion of comments, and linear flow of program control.

## 1.6 Procedures and Data

Procedures and data have been recognized to be separate entities of programs with distinct properties. Data objects may change their values dynamically, whereas a procedure should be static and immutable. Programs which modify themselves are difficult to comprehend and virtually impossible to verify.

## 1.7 Language Size

Language size as measured by the number of key words or language concepts has been seen to be important. Relatively small languages are easier to implement and make it possible for the programmer to gain complete mastery of the programming language. The language designer must carefully synthesize a limited number of features from among the vast number of potential choices.

## 1.8 Analyzability

Analyzability is measured by the level of support provided by language features that simplify the process of analyzing already-written programs by automatic methods. (This is basically a "recognition" as opposed to a "compilation" problem, since one can assume that the language's compiler provides comprehensive syntax analyses.) In a sense, this characteristic crosses all the boundaries outlined above, since analyzability cannot be achieved without significantly good "scores" on other factors. Included in this characteristic is the ease with which the language can incorporate semi-active assertions and/or instrumentation of various kinds. In addition, the simplicity of symbolic analysis/execution/evaluation processes that would operate on programs written in the language is also a factor.

# SECTION 4. CRITERIA FOR TINMAN "BASE LANGUAGE" SELECTION

1. INTRODUCTORY NOTES

The outline below represents a proposed statement of the criteria for evaluation of a candidate language to "represent" the TINMAN. Ultimately there are four possible routes to choosing an appropriate base language:

1. Design a new language that meets the TINMAN specifications
2. Change TINMAN to match an existing language.
3. Change an existing language to meet TINMAN.
4. Compromise between an extended existing language and a reduced TINMAN.

Regardless of the route taken, the following set of criteria is essential for evaluation of a candidate language.

2. LANGUAGE DESIGN AND DEFINITION

- 2.1 Language Style
  - 2.1.1 Programmer acceptability
  - 2.1.2 Major Issues:
    - a. uniform referents
    - b. abstract types
    - c. extensibility
    - d. "nearness" to TINMAN
  - 2.1.3 Secondary Issues (Details)
- 2.2 Language Syntax
  - 2.2.1 Programmer acceptability
  - 2.2.2 Syntax Modification/Extension Cost, Time, Risk
  - 2.2.3 Psychological Factors
- 2.3 Language Semantics
  - 2.3.1 Informal description
  - 2.3.2 Semi-formal Description
  - 2.3.3 Formal Description
- 2.4 Tutorial Aspects
  - 2.4.1 Learning Curve
  - 2.4.2 Unlearning Curve (If applicable)
  - 2.4.3 Tutorial Subset--its clarity

3. COMPILER SPECIFICATION AND DESIGN

- 3.1 Assessment of Compilability of Language
  - 3.1.1 Modification of existing compiler
  - 3.1.2 Generation of new compiler
  - 3.1.3 Run-time support needed

3.1.4 Size of Syntax Processed
3.1.5 Diagnosibility
3.1.6 Efficient Code/Optimization
3.1.7 Reformatability
a. expansion
b. reference
c. other details

3.2 Design of Compiler Specification
3.3 Detailed Compilation Algorithm Analysis
3.4 High-level Compiler Design

4. COMPILER IMPLEMENTATION

4.1 Self-Boot
4.2 Production Version
4.3 Optimizing Capability
4.4 Debugging Capability
4.5 Interactive Capability
4.6 Library Capability
4.7 Program Analysis Capability
4.8 Other Capabilities

5. LANGUAGE CONTROL AND MAINTENANCE

5.1 *Existing Dispersion*
5.2 Controllability
5.3 Extension of Control to Subsidiary Features Application (libraries, etc.)
5.4 Validation Program Techniques

6. COMMENTARY

The above outline was produced on the assumption that the end result of the efforts of the HOLWG (perhaps in late 1977 or 1978) would be the design and definition of a new programming language DoD/1. It attempts to cover a number of the issues that must be addressed before such a language can successfully achieve the goals to which the entire HOLWG effort has been directed.

It has been recognized that the creation of a language which meets the TINMAN (or some future set of) requirements is not sufficient to guarantee the success of this effort. There are a number of concomitant factors which are equally important, including the following:

1. There must be effective mechanisms for teaching the language to programmers. The cost of instruction and the preparation of tutorial materials, combined with the loss of productivity while programmers achieve proficiency with a new language, is comparable in cost to the language design and implementation effort itself.

2. Accordingly, language features in a new language should resemble features in existing languages when those features serve a similar purpose and should differ when different purposes are to be served. The language should not be burdened with complicated or inconsistent rules of syntax, since they will be a source of programming errors and will be easily forgotten by programmers.

3. The value of the HOLWG effort is in the creation of a standard programming language which supports programming discipline and the creation of high quality software. Any activity which tends to dilute the standard must be strongly discouraged.

4. As a result, the language should be complete enough that there will be no need to extend the base language, except to develop libraries of procedures and abstract data types.

5. There should be a tutorial subset of the language, but no subset implementations. In other words, a programmer should not necessarily be taught the entire language initially, even though the language processors with which he/she will work will support the entire language. Furthermore, the tutorial materials developed for DoD/1 should present the language in the context of recent efforts in programming methodology and good programming style in order to motivate language conversion.

6. Compilers should be built with portability as an objective so as to encourage the availability of the language on a wide variety of machines. It may be valuable to develop an intermediate language such as OCODE used by BCPL in order to simplify portability.

7. Validation and testing mechanisms will have to be created for DoD/1 along the lines of the mechanisms which are in place for COBOL compiler testing validation. In addition to validating correct

programs, attention should also be given to the behavior of the compiler in the presence of incorrect programs. The validation software will need to be extremely rigorous due to the requirement concerning no subsets or supersets.

8. The HOLWG, an X3 committee of ANSI, or an equivalent organization will have to maintain responsibility for the language and its development over time.

There are a number of steps that must be taken in order to assure that the effort of the HOLWG will be successful. It is recommended that the HOLWG include a major evaluation component within the language design effort to make certain that the language designed and the implementations constructed meet the objectives.

Lastly, it is recommended that a number of generic program examples be specified by the HOLWG to be used to demonstrate the ease and comprehensibility of each candidate programming language. It is envisages that a set of small problems, each designed to illustrate the way the language handles a critical issue, would be created. The HOLWG would require code, written in the candidate language representing the solution to each problem, to be submitted by the language designer, thereby facilitating practical comparison of the way different proposed languages handled a variety of problems.

# CHAPTER 3
# TINMAN

The contents of this chapter of the report involve material relating to assessments of the TINMAN itself.

## Section 1. Comments on the TINMAN Requirements

# SECTION 1. COMMENTS ON THE TINMAN REQUIREMENTS

## 1. INTRODUCTION

The TINMAN represents an idealized set of language design goals, intended to bring together all of the recent advances in programming languages and all of the intriguing concepts in programming languages into a single language. What is envisioned is a single language (the universal computer language again?) which supports all of the key concepts of abstraction and modularity on one hand and gives the programmer control over the low-level details of object representation and execution regimes on the other hand. In addition, TINMAN envisions creating language control mechanisms by which a single organization (presumably some DoD agency) would oversee both the design and control of the language and the construction and management of all translators for all machines. This would involve an elaborate validation effort to ascertain that the implemented language conformed exactly to the language specification, with neither extensions nor omissions.

In examining the specific requirements more closely, it can be seen that some of them are unachievable in practical languages, while others are simply self-contradictory. Furthermore, the language which results from attempting to meet all of the TINMAN requirements is of necessity large and complex. The result of such a design is a language which is difficult to learn and remember, as well as being somewhat difficult to implement. This language would present a large number of concepts which are new to the average programmer as well.

This section examines some of the TINMAN requirements in more detail, paying particular attention to those objectives which appear to be unrealistic or contradictory. The intent is either to suggest modifications to TINMAN for use in the next evaluation/design phase or to suggest relaxations in the

requirements that should be used to evaluate existing languages against the TINMAN. The result of this set of recommendations is a somewhat smaller language which gives more attention to the issues of abstraction, modularity, control structures, and program readability, with less attention being given to the lower level details.

## 2. ON HIGH LEVEL LANGUAGES

The basic concept of a high-level language is to provide the programmer with a set of facilities which are meaningful in the terms of the problem being solved rather than in terms of the machine on which it is being solved. Early research in high-level languages confirmed that programmers were more productive and produced higher quality code when working with a high-level programming language. The use of algebraic expressions in FORTRAN or record structures in COBOL was far more meaningful to the programmer than was a set of machine language (or even macro assembly language) instructions.

Recent work in programming methodology has gone on to validate that concept still further. Abstract data types remove still another level of object presentation throughout much of a program. The programming language is seen as an extension of the problem-solving process; it is desired that the transformation from a program design (problem solution) to a correctly functioning program be simplified as much as possible.

The value of abstraction is to hide the lower level details of implementation from the programmer. The value of modularity is to manage the intellectual complexity of the programming task, to make it feasible to divide a large project into pieces, and to permit parts of a program to be isolated in such a way as to minimize side effects and the problems of connecting modules.

The goals of efficiency have generally been regarded as, at least partly, secondary to the goals of abstraction and modularity, even though it is recognized that a poor compiler may lead to much slower program execution. Inefficient execution-time performance tends to eliminate a language from actual use. Rather than giving the programmer a great deal of power with respect to storage allocation or object code, languages have been restricted in such a way as to insure that efficient object code can be generated by an optimizing compiler.

## 3. ABSTRACTION VS. EFFICIENCY IN TINMAN

There is an inherent conflict between the goals of efficiency and abstraction. TINMAN proposes to achieve both, and some significant problems result.

The requirements which address efficiency directly are the following:

B6. "... The operations 'and' and 'or' on scalars will be evaluated in short circuit mode."

C4. "...constant expressions will be evaluated before run time."

I2. "... The programmer will be able to override these defaults when necessary."

J1. "The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs."

J2. "Any optimizations performed by the translator will not change the effect of the program."

J3. "The source language will provide encapsulated access to machine dependent hardware facilities including machine code insertions."

J4. "It will be possible within the source language to specify the object presentation of composite data structures..."

J5. "The programmer will be able to specify whether calls on a routine are to have open or closed implementations..."

L3. "The translator will minimize compile time costs..."

## 3.1 Compile Time Costs

Requirement L3, stressing minimization of compile-time costs, seems especially problematical. First, requirement L6 states that the compiler should perform as much checking as possible, including type checking and verification of semantic restrictions. Second, the size of the proposed language is such that construction of an efficient translator is nontrivial. Third, minimization of compile-time costs seems contradictory to objectives of execution-time efficiency, since many of those gains come as a result of optimizations made at compile-time, often requiring multiple passes over the intermediate code or initial object code in order to generate efficient code. If one wishes to meet J1, for example, all DO loops (or their equivalent) must be checked to see when the step and termination values are constant and when they are variable (unless the language forbids this use of variable number of iterations). Fourth, the availability of compile-time facilities, including conditional compilation, represent an additional burden on the compiler, increasing compile time costs. Fifth, it is expected that most uses of programs will be in operation rather than in development; it would appear that the goal of efficiency would dictate that the effort be directed toward the generation of efficient executable code rather than toward minimization of compilation costs.

## 3.2 Optimization

The optimization requirements present a similar set of problems. J2 is particularly difficult, especially when viewed against C1: "Side effects which are dependent on

the evaluation order among the arguments of an expression will be evaluated left-to-right." C1 restricts a significant number of potential optimizations, including the collateral elaboration notion of ALGOL 68 and the grouping by hierarchy of LIS, whereby the multiplication would be performed first in A + B*C. It is recognized that, particularly for floating point numbers, any change in ordering of evaluation of an expression can change the result, due to round-off errors in the least significant decimal places. A + B - C may easily yield a different result than A - C + B, particularly if this sum is performed repeatedly.

In short, the combination of J2 and L3 would seem to indicate that no optimization whatsoever should be performed on programs, despite the fact that achievement of J1 requires such optimization.

Requirement J4 presents additional complications for the compiler. The ability of the programmer to specify the object representation of a composite data structure is very unpleasant. If there were severe main memory requirements for an application, the programmer might very well pack the data objects tightly as possible. The object code that would be required to extract the appropriate bits for each item of a composite data structure would impose considerable overhead at both compilation and execution time.

It is recognized that TINMAN wishes to give the programmer the ability to make the time/space efficiency tradeoff in programs, but the other TINMAN requirements then run counter to that goal.

## 3.3 Language Extensions

A key question that was raised by Standish and others with respect to extensible languages was whether programmers are able and qualified to make these language

extensions. This question will certainly arise in conjunction with the language derived from these requirements, and is also applicable to whether the programmer is best qualified to choose object representations.

Perhaps a higher level directive should indicate available space, with the compiler or optimizer trying to fit a program into a space by changing object representations if the compiled object code is too large. There is the danger that the programmer will go to considerable effort to optimize space utilization when space is not an issue at all for a given application.

### 3.4 Expression Evaluation

The use of short circuit mode for evaluation of 'and' and 'or' operations is a potential source of errors should programmers include functions having side effects within the Boolean expression to be evaluated. More complicated object code is needed to achieve the short circuiting of execution than is required to simply evaluate the Boolean expression. Again, the requirement for minimized compilation is contradicted by another TINMAN requirement. It is recognized that programmers must be taught not to program in such a way that side effects are created in expressions which are to be evaluated. If this is done, then there is no need for the requirement about side effects (C1). In short, performing short circuit evaluation of expressions according to B6 and then permitting side effects (C1), even if the subexpressions are evaluated left-to-right, is a likely source of programming errors.

### 3.5 Direct Code

TINMAN seems somewhat schizoid on the distinction between supporting high-order language constructs and giving the programmer all of the power of the machine language

programmer. Perhaps this is because TINMAN does not really address the set of applications to which it is intended. If the envisioned language is to be a system implementation language (like BLISS, LIS, etc.), then the machine language and data representation requirements may be needed; if it is to be an applications language for commercial data processing (like COBOL), then all of these low level issues should be removed in favor of presenting a clear set of abstractions to the programmer. The language simply cannot be all things to all people--efficient object code, efficient space utilization, minimal compilation time, suitable for abstraction and modularity, and so on.

## 4. HIDING OF TYPES

The TINMAN envisions a language with a powerful set of data types with a facility for introducing new data types, new operations on those types which are encapsulated with the type definition, and the ability to extend base language operations to the new types. The requirements are very clear about the use of types in the language, with emphasis on clarity and consistency. Yet there are some problems.

One wants to know the type of the object with which one is dealing and should be extremely careful about the mixing of data types. However, some of the requirements tend to blur this distinction. This comes about in several ways.

First, the equality comparison is supposed to be applicable to any two objects, regardless of type. It is not clear why this should be. What is the meaning of comparing a real number with a set, for example, or an integer with a record? A more useful requirement would be that the equality comparison would be applicable to two objects of the same type or to two objects of united type (using the ALGOL 68 term), whose intersection types are not null. If one variable was MODE A = UNION (INTEGER, REAL) and the other was MODE B = UNION (INTEGER, CHARACTER), it would make sense to compare them as integers,

for example. In general, programmers should have a consistent use of variables of types so that meaningless comparisons do not occur, rather than permitting the comparison operator to be universally applied. Another meaningful extension would be to require the comparison operator to be defined for all new programmer-defined data types (as is done in CS-4).

Second, assignment of conformable arrays and transfers between records or arrays of identical logical structure is a useful shorthand notation, particularly for records, but obscures the fact that the assignment applies to a complex data structure, thereby detracting from program comprehensibility and readability.

The object of providing a uniform reference notation is difficult and elusive. On the one hand, one would like to be consistent, but on the other hand, one would like to know something about the type of the object on which an operation is being performed. PASCAL, for example, has different reference notions for arrays, procedures, records, and pointer references. It is immediately apparent as to which type of object is being used. Geschke and Mitchell [18] have examined the problem and concluded that, while one could achieve this objective, it may not be worth doing so.

Fourth, the optional specification of parameters on the formal side of a procedure declaration (C8) does not seem to be the best way to deal with generic procedures, especially since one wants to constrain the operations that are performed on objects of various types. Even PL/I has a more intelligent solution. One may specify a parameter to be a united type, e.g., MODE A = (REAL, INT, STRING, BOOLEAN, ...) so that the procedure call can accept any parameter which is of one of those types. This provides the facility desired by C8.

## 5. NUMERIC ARITHMETIC

TINMAN requirements specify both fixed and floating point real arithmetic, despite the fact that fixed-point arithmetic

is the source of a significant number of execution time errors. It would appear that fixed point arithmetic is desired to avoid the potential difficulties of roundoff error associated with floating point computation, when carried out in binary. Yet fixed point arithmetic has the danger of truncating the most significant digits in arithmetic calculations, of losing signficant digits on both the high order and low order end, and adds several additional requirements to the language.

Much of what is required in fixed point arithmetic could be handled with integers; the remainder could almost certainly be handled by floating point arithmetic where the calculation is carried out in decimal rather than in binary. Binary can be considered the default case. This removes one type from the language, as well as a common source of execution-time errors. This would again provide the programmer with the choice between efficiency and space.

## 6. DATA STRUCTURES

The TINMAN goes to some lengths to propose a set of data objects as built-in types which provide a firm basis for the definition of new types and for the implementation of recursive data structures as discussed by Hoare. Hoare's contention is that the defined set of data structures makes use of the pointer unnecessary, even if its use is restricted. If one believes in recursive data structures, then pointers and pointer mechanisms should not be a requirement in the language.

## 7. THE GO TO

A similar situation applies for the GO TO. The TINMAN requirements go to some length to provide a powerful set of control structures, including exception-handling and loop termination from any point within a loop, the two situations for which the GO TO is now used most commonly. Yet TINMAN then turns around and requires a GO TO statement after rendering it

totally unnecessary, giving the flimsy excuse that it is needed for transliterating programs written in older languages. Again, TINMAN seems of two minds, trying to decide between taking a very modern, high level approach to programming, and retaining the old ways which led us into the present state of affairs.

8. EXTENSION OF OPERATORS

ALGOL 68 provides a particularly elegant way of extending existing dyadic operators to new data types in the language. One can envision adding disparate structs together given the appropriate extension of the + operator. There is also the ability to define new dyadic operators, such as $$, to mean anything at all. In addition, TINMAN specifies that there must be a mechanism for associating an operator hierarchy with the new operators. ALGOL 68 includes the prio facility which permits this to be done. Unfortunately, prio can be applied to existing operators on built-in types, violating H2. It is not immediately clear how to make new data types indistinguishable from old types (E2), unless the operator priority for existing operators is fixed regardless of the type to which it is applied. If new infix operators are permitted for the new data types, precedence must be defined (in violation of H2?). In short, there are some difficulties in this area which must be resolved.

9. PROGRAMMABILITY

Although much harder to support by cogent and specific argument, the issue of programmability for the language envisioned by TINMAN is important to discuss. Programmability refers to the "naturalness" of the language from at least two viewpoints: learning to use the language, and remembering what it has available.

The normal pedagogical approach taken in informing even an expert programmer about a new language involves absorbing complete information about a subset of language. This technique must ultimately be applicable to all persons interested in learning

to use the language. TINMAN seems to envision a language of such intensity, flexibility, and complexity that one wonders whether there is a straightforward method for teaching the rudiments without overburdening the student with unwanted detail. The situation is reminiscent of some constructions in JOVIAL which on first encounter are arcane, to say the least.

It must be recognized that 90% of programs will use a very limited percentage of the language power. Will some of the sophisticated features of the language get in the way of average programmers? It should be clear that if a language like that envisioned by TINMAN intimidates everyone, then the language will not achieve the goals of the HOLWG effort.

10. ANALYZABILITY

Software analysis tools, as they are currently constructed, and as they can be projected from present technology, operate on the notion of having database representation of the (assumed syntactically correct) program in its entirety. Picture something like an "exploded parts diagram," with the entities (variables, operations, sentences/statements, blocks, etc.) as the "parts."

Once this information is available (a task accomplished by syntactic recognizers operating over the database), one can run algorithms that do many interesting and valuable things. Typically these involve functions that are only cost-effective to perform infrequently (otherwise, they would be incorporated in the compiler), or are valuable only when performed on a complete (or near-complete) software system. Acceptance testing, ferreting out hard-to-find errors, verification, control flow analyses, and quality/style investigations are some examples.

How does a TINMAN induced language simplify (or complicate) performing these kinds of functions? The answers lie in considering the rate of growth in complexity of the needed database; this is equivalent to assessing the density of semantic connectness within and among programs written in the TINMAN advocated language.

Notwithstanding some of the evidently contradictory requirements of TINMAN, the abstraction capability requirement seems to be the biggest difficulty. Consider the problems encountered with a software system which is completely expressed in user-defined abstract types. In this case (which is probably unlikely in practice and is, also, probably the worst possible situation), nothing in the program can be recognizable to the checking algorithms.

The solution to this problem, of course, is to develop algorithms that can derive the level of understanding required from the abstract definitions or which can gain access to the lower level definitions. But to achieve this may take many years after a language finds widespread use; consider that it was two decades between the introduction of the first HOLs and the development of automated program analyzers that could check coherence of programs written in them.

## 11. MISCELLANEOUS ISSUES

There are a few other points that should be cleared up. The requirement prohibiting nonrecursive routines embedded in recursive routines is reasonably trivial to achieve, but doesn't prevent some of the execution time problems associated with recursion, thereby contradicting efficiency considerations. This is still another place where the TINMAN seemed to be of two minds: the FORTRAN/COBOL mind which banishes recursion because of the required runtime support, and the ALGOL/PASCAL/LISP mind which sees the elegance of recursion despite the overhead. The present restriction (G5) is artificial.

The use of a break character in numeric literals seems to present certain implementation difficulties ("just another thing to check") and might well be dropped, particularly since numeric literals are not frequently given to more than 8 places of precision.

The issue of numeric precision for reals and integers should be thought through a bit more fully. They provide the ability to specify a representation by indicating the number of places of precision required, whether or not that precision is meaningful in the context of the computation. The experience with PL/1 has shown that serious errors can result unexpectedly from the practice of operating on numbers with different precision. This is particularly true when mixing fixed point quantities with floating point, and was the motivation concerning the earlier suggestion about eliminating fixed point artihmetic. (The problem here is the universal computer language problem--the COBOL programmers like fixed point, but it isn't needed anywhere else.)

The overhead associated with providing a variable number of arguments is considerable and goes against most language features available to programmers. One might define a certain number of built-in functions or procedures which accept a variable number of arguments, e.g., MAX, but it is not apparent how this facility might be given in general for the definition of functions/procedures by programmers.

The TINMAN requirements concerning comments indicate that the language should be both free-format and line-oriented, so that one doesn't run into the problems connected with failure to terminate a comment properly. While this can be handled satisfactorily, it presents some additional minor difficulties to the compilers which attempt to "prettyprint" the program text (as in LIS).

## 12. SUMMARY

These comments have been directed to improving the consistency of the TINMAN requirement and to suggesting

some areas in which TINMAN requirements might be tightened or loosened so that a smaller language will result. Such a language would not provide some of the machine dependent facilities of the present set of requirements, and in fact, would actively discourage such programming style since those techniques decrease program portability and understandability.

They were motivated in part by examination of CS-4, a language which is being dynamically designed to meet the TINMAN requirements. CS-4 seems unwieldy as a programming language in several respects (taken up in the subsection on CS-4 vs. TINMAN), leading one to believe that TINMAN may be asking for too much in an inconsistent way.

Given these suggestions, it should be possible to design a language (or modify an existing one) in such a way that it meets the criteria for high level languages and so that it can be used effectively by programmers.

# CHAPTER 4
# ALGOL 60

The contents of this chapter of the report involve material relating to ALGOL 60.

Section 1. Evaluation of ALGOL 60 Against Language Design Criteria

Section 2. Evaluation of ALGOL 60 Against the TINMAN

Section 3. Language Evaluation Summary for ALGOL 60

Section 4. Detailed Comparison of ALGOL 60 to the TINMAN

# SECTION 1. EVALUATION OF ALGOL 60 AGAINST LANGUAGE DESIGN CRITERIA

## 1. INTRODUCTION

This section examines the extent to which ALGOL 60 satisfies the criteria for a "good" higher-order language as presented in "Criteria for Evaluation of High-Order Languages." ALGOL 60 is seen to fulfill the requirements moderately well. However, there are significant weaknesses in most of the areas, thereby making it unsuitable for further consideration as a base language for the TINMAN.

## 2. ABSTRACTION

ALGOL 60 provides facilities for procedural abstraction through functions and procedures. Procedures and functions accept two kinds of procedures: those called by name and those called by value. Parameters passed by name are evaluated on each use of the parameter and may change dynamically. If a function or procedure name is passed by name to another or function, that function will be repeatedly called and evaluated. An expression will similarly be evaluated repeatedly; this is known as "Jensen's device." However, ALGOL 60 functions can only return values of type real, integer, and Boolean.

ALGOL 60's facilities for data abstraction are very weak, by comparison with other languages. ALGOL 60 was conceived as a language for numerical scientific programming applications and provides only the three data types of real, integer, and Boolean; there is no mechanism for defining either new types or new operations on existing types. Although there are numerous extensions to ALGOL 60 which provide more sophisticated capabilities, they are not part of the Reference Report. There are no records; however, there are arrays of the three basic required types.

## 3. MODULARITY

ALGOL 60 is a block structured language in which variables may be passed between blocks. A procedure or a function with

local declarations is considered a block. As noted above, there are parameter passing mechanisms between procedures and functions. Variables which are declared local to a block are known to that block and to all enclosed blocks, with no way of limiting that access, except for renaming the variable in an inner block. A programmer must be careful in the use of variables which are global to other blocks.

In general, it is possible to achieve modularity in ALGOL 60 programs by careful use of variables and well-constrained standards for parameter passing. The availability of global variables with the ability to cause side effects in procedures and/or functions can be a technique to subvert the goals of modularity. In summary, modular programming may be accomplished in ALGOL 60, but only if the programmer takes care.

## 4. LINEAR FLOW OF CONTROL

Control structures in ALGOL 60 include the _if-then-else_, the _switch_, the _for-step-until-do_, and the _for while_, in addition to the _go to_. While these control structures, supplemented by procedure calling mechanisms (including recursion) are sufficient to eliminate the need for the _go to_, in practice they leave something to be desired. Relatively few ALGOL 60 programmers use Boolean variables with the FOR-WHILE construct to terminate loops. The tendency is to use to _go to_.

Neither the _go to_ or the _switch_ have adequate constraints on branching. Both permit transfers of control to any program location, either forward or backward in the same or a _surrounding_ block. A beter solution would constrain the switch to a group of statements as with the case statement in PASCAL, and constrain the _go to_ to _forward_ branching in the same local scope of control, i.e., the same block. Although it is possible to write programs with linear flow of control, the programmer must make an extra effort to do so.

## 5. CONTROL OF SCOPE AND BINDING

ALGOL 60 handles this TINMAN requirement quite well. Variables have an explicit scoping defined by the point of declaration. With variables declared to be of type own, there is a distinction between scope of access and scope of allocation, since such variables exist from the point of initial allocation upon initial entry to their block until program termination. However, there is no mechanism for preventing access to a variable in an inner block, except for redeclaring the same variable (possibly, with different attributes) which creates a new variable with the same name. This makes the original instance of the variable inaccessible within the block and any of its enclosed blocks.

Variables are bound to a single type at their declaration point. There are no union types and there is no way for a variable to assume a different type. Actual and formal parameters for procedures and functions must also agree in number and type.

## 6. READABILITY AND COMPREHENSIBILITY

ALGOL programs tend to be quite understandable when they are dealing with numerical scientific problems. For other applications, the programmer must be careful to select meaningful variable names (identifiers), use Boolean procedures, and isolate those routines which use the actual representation of a data object rather than its abstraction.

ALGOL 60 has been used for compilers, text editors, and many other programs which do not involve scientific computation. The limited number of control structures, the block structure, the full declaration of variables and the commenting facility all contribute to program comprehensibility. ALGOL 60 is a simple enough language that most programmers feel comfortable with it and can easily understand the intent of well-written programs. (As with all other languages, one can write poor ALGOL 60 code, but the language does not pose the intrinsic readability problems of languages such as LISP or APL.)

## 7. PROCEDURES AND DATA

ALGOL 60 makes a firm distinction between procedures and data. There is no mechanism which permits data to be executed as program text. Variables must be declared, so it is immediately apparent as to what is program text and what is data. ALGOL 60 programs which use linear control flow are thus verifiable and compilable.

## 8. LANGUAGE SIZE

By most common measures, ALGOL 60 is a relatively small language. It contains only 22 key words, less than FORTRAN or any other widely used language (BASIC excepted). The syntax is described in well under 200 productions in BNF and the complete ALGOL 60 report occupies fewer than 40 printed pages. ALGOL 60 is suitable for teaching to beginning programmers, who are able to grasp virtually all of the language concepts in a relatively short period of time.

The relatively small language size is not directly reflected in the size of ALGOL 60 compilers, however. The ALGOL 60 block structure, the two forms of parameter passing, and the availability of recursion necessitate a certain degree of run time support and a more sophisticated symbol table mechanism. Certain ALGOL 60 features, including dynamic own arrays and integer labels, have generally not been implemented because of the problems caused by these features.

## 9. ANALYZABILITY

ALGOL 60 programs lend themselves to analysis quite well, since various blocks, functions, and procedures may be instrumented without interfering with the rest of the program. One must be careful in the methods used for exiting those blocks which are being examined, so that measurements are accurate.

ALGOL 60 was also the first language in which concepts of program proving were applied. Much of the research into

programming methodology has originated with ideas which led to higher quality ALGOL 60 programs, i.e, programs that were easier to prove.

## 10. SUMMARY

The criteria used above are fundamental to language design and underlie much of the current research into programming languages and their design. It is generally accepted that the first two criteria, abstraction and modularity, are the areas in which the most important research efforts are currently being made. The shortcomings of ALGOL 60 in those respects, particularly in its limitations concerning data types, detract significantly from its acceptability.

Despite ALGOL 60's poor showing on the TINMAN criteria, any language resulting from the DoD HOL Project will undoubtedly exhibit many of the features of ALGOL 60 and its descendants. Virtually, all of the candidate langauges being studied owe a sizeable portion of their design to concepts introduced with or built upon ALGOL 60.

# SECTION 2. EVALUATION OF ALGOL 60 AGAINST THE TINMAN

## 1. OVERVIEW

The ALGOL 60 language introduced a large number of important concepts for programming languages, including such notions as definition of all variables, block structure, formal language definition, parameter passing by name and by value, and several other features. Its impact on other languages and on programming language research cannot be overstated.

Since the publication of the ALGOL 60 Report, a vast number of new languages have been developed, many incorporating a number of new concepts related to programming languages. Among these are modularity, abstract data types, restricted control structures, and record-type structures. The lack of input/output facilities for ALGOL 60 prevented the language from receiving wide acceptance and use.

The TINMAN recognizes a number of these advances as being fundamental to a "modern" programming language. Accordingly, ALGOL 60, despite its elegance, and despite the recent proposed modifications to ALGOL 60, fails to measure up to TINMAN. In fact, it falls so far short that it is not really necessary to give serious consideration to ALGOL 60 as a basis for TINMAN, since other languages have built upon the strong foundations of ALGOL while adding some of the other important notions.

The following discussion, then, is an abbreviated analysis of ALGOL 60, with emphasis on the shortcomings of the language with respect to TINMAN.

## 2. DATA AND TYPES

While ALGOL 60 is a fully typed language which supports n-dimensional arrays, it falls short on virtually all of the TINMAN specifications in this area, beyond A1. There

are no fixed point reals and no records. With respect to characters, there is a string data type, but there are no facilities for manipulating strings. There is no means for specifying the precision of fixed or floating point reals (A3). The absence of fixed point numbers rules out A4. There are no enumeration types (A5) in the sense of PASCAL scalar types. ALGOL 60 has dynamic array bounds. Both the lower and upper bounds for an array may be dynamically determined at run time upon entry to a block, violating A6. There are no records, so that there is no facility for alternative structures as with variant cases of RECORDS in PASCAL or PLEXES in LIS, so that A7 is not met.

3. OPERATIONS

Assignment and reference operations are defined for all built in data types. There are no extended data types and only a limited number of built in types (real, integer, Boolean). There are no union types (B1). It is, in general, not possible to compare two data objects (regardless of type) for identity. A Boolean cannot be compared to a real or an integer, violating B3. Reals and integers may be compared; the comparison is done in real arithmetic so that errors may result as the outcome of the conversion. Relational operations are defined for reals and integers. A different set of operations (those of propositional logic) are defined for Booleans. B4 is satisfied for integers and reals; note that there are no fixed point quantities. B5 is also met. There is no "nor" logical operation. "And" and "or" are carried out fully; any side effects are carried out--short circuits are not made. There are no scalar operations on comformable arrays. All array references must indicate the proper number of subscripts, except in the declaration of an array as a formal parameter.

Implicit type conversions are limited. Entier is a real-to-integer conversion function, but rounding is done automatically if a real quantity is assigned to an integer. The absence of exception conditions rules out B9. The absence of input/output facilities rules out B10; even the "New" ALGOL 60 Report [17] does not fully meet this requirement. Finally, there are no power sets (B11).

## 4. EXPRESSIONS AND PARAMETERS

ALGOL 60 expressions are evaluated left-to-right; thus, any side effects will be treated consistently. ALGOL 60 has 9 levels of operator hierarchy. There is occasional confusion to the reader of a program as to the levels of precedence should an expression mix logical, relational, and arithmetic operations. ALGOL 60 does not support constants (C4). The parameter rules are relatively consistent for procedures; there are no exception handling or parallel processing facilities, so that all parameters are to and from procedures. These are passed by name or by value. Some other problems in this area are discussed below.

Formal and actual parameters must agree in type. If an array is passed as a parameter, it is only necessary to declare the array and its type in the procedure heading, without declaring the number of dimensions or the size. However, the subscripts must be shown within the procedure body, so that the number of subscripts must be determinable at compile time. Thus C6 is satisfied.

ALGOL 60 has two kinds of parameters. Exception handling parameters are nonexistent. Procedure names may be passed as parameters in much the same way as other parameters. It is possible to cause some unexpected side effects by passing a constant where a value parameter is expected and then making an assignment to the formal value parameter; this is

not properly handled by all ALGOL 60 compilers. Parameter passing by name is different from procedure passing by reference. When a parameter is passed by name, it gives the ability to pass a procedure name and effect Jensen's device. The TINMAN specification wants parameters to be passed by reference and by value so as to clear up the confusion when an expression is passed by reference/name. That way, passing a procedure name must be handled by a separate class of parameters.

There are no facilities for generic procedures or for type unions. The type of all formal parameters must be specified, contradicting C8. The number of formal and actual parameters is constant and they must agree in number, violating C9.

5. VARIABLES, LITERALS, AND CONSTANTS

ALGOL 60 has no mechanism for declaring constants, nor a method for initialization of values (D1, D3). D4 is not met because of the absence of fixed point types. D5 is not met fully since there is no way to specify a range of values associated with a type, nor is there a way to define a type. D6 is not met since there is no pointer mechanism. Only D2 is met, since there is a consistent interpretation for the use of literals for Booleans and numerics.

6. DEFINITION FACILITIES

None of the definition facilities requirements are met. The user does not have the ability to define new data types (E1), except insofar as an array can be considered a new data type. This fact means that E2, E4, E5, E6, E7, and E8 are not met. E3 is satisfied, however.

7. SCOPES AND LIBRARIES

There are minor distinctions between scope of allocation and scope of access. One may declare an <u>own</u> variable,

which continues to exist from the time of first entry into the block in which it is created until the termination of the program so that the value is preserved upon subsequent entry into that block. There are some difficulties with the concept of dynamic own variables, i.e., arrays which could change their bounds. Most ALGOL 60 implementations do not support this feature and the set of modifications defined in 1976 abolishes this feature. The scope of identifiers may be determined at compile time, satisfying F3.

There are no facilities for limiting access to structures (F2).

The language provides a set of nine built-in functions, which may be viewed as comprising the beginning of a library. However, the existence of libraries is implementation dependent. Since there is no standard input/output facility, any mechanism for storing a pool of data must be machine dependent (F4, F5, F6). The lack of I/O eliminates F7.

## 8. CONTROL STRUCTURES

The ALGOL 60 control structures are relatively limited. G1 is partially satisfied with mechanisms for sequential, conditional, iterative, and recursive control. However, there are no facilities for parallelism, exception handling, or asynchronous interrupt handling. G2 is partially satisfied--the language has a go to statement, but labels may be anywhere in the program and there are few restrictions on branching. Furthermore, there is some problem with the use of numeric labels; some syntactic ambiguities arise when they are used as parameters to procedures. The modified ALGOL 60 Report of 1976 eliminates them. The conditional control structures are the IF-THEN-ELSE and the SWITCH. IF-THEN-ELSE is not fully bracketed and nested IF-THEN-ELSE statements must be enclosed by BEGIN-END to prevent the

ambiguity of a "dangling ELSE."

The rules concerning iterative control structures are met, since the control variable may be a normal variable whose name (but not its value) is accessible upon exhaustion of the loop. (Both the name and the value are available upon termination of the loop from within.) There is no mechanism for ending a loop other than upon completion of its scope, i.e., no embedded termination condition. Entry is restricted to the top of the loop. The step part of the FOR statement can be executed with each iteration of the loop, which can cause excessive overhead even in simple cases.

Recursive routines are permitted; there is no restriction on the declaration of internal non-recursive procedures internal to the recursive procedure. The absence of parallel processing, asynchronous interrupt handling, and exception handling eliminates G7, G7, and G8.

9. SYNTAX AND COMMENT CONVENTIONS

ALGOL 60 meets H1 fully. ALGOL 60 was probably the first programming language which achieved this objective; ALGOL 60 caused people to recognize the value of mnemonic variable names, uniform context-free grammars, and easily recognized forms. ALGOL 60 also meets the TINMAN requirements H2 concerning inability to modify the source language syntax, H3 concerning the ability to implement the language with a 64 character subset, H6 concerning the reservation of keywords, and H8 concerning matching of parentheses.

H4 is almost met, but there are no provisions for break characters in either identifiers or literals. Lexical units cannot, in general, be continued across lines, although some compilers treat a program as a continuous stream with no line dividers. End of line is meaningless for ALGOL 60, since there is no input/output(H5).

ALGOL 60 has a uniform comment convention, beginning with the symbol comment and ending with a semicolon ; . This is not determined by source lines, since that is not a meaningful ALGOL 60 concept.

The uniform referent criterion is not met. Procedure and function calls use parentheses, while array references use brackets.

H10 is satisfied, since the use of symbols is consistent through the language.

## 10. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

I1 is satisfied because decisions affecting program logic are explicit. The precision of reals and integers is handled by default with no mechanism for overriding them (I2). There are no compile-time facilities associated with ALGOL 60 (I3, I4).

I5 is satisfied in that the source language contains a clearly identifiable base language.

I6 is not satisfied since there is no provision for specifying language restrictions in the language definition. However, I7 is satisfied, since no language restrictions are built into the language definition. The meaning of the second part of I7 is not clear: "Language restrictions which are inherently dependent on the object environment will not be built into the ... translator."

## 11. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

ALGOL 60 does not provide any machine dependent facilities or access to machine dependent facilities within the language. There is no way to specify object representations. The efficiency of code generated by ALGOL 60 compilers varies drastically; there are some potential

overhead costs associated with the language. For example, the FOR statement is completely general in that the control variable, the step size, and the termination values may be changed within the scope of the loop. Furthermore, the step size and the termination point may be expressions involving function calls so that they must be evaluated through every iteration. Even the FOR statement

```
FOR I := 1 STEP 1 UNTIL 10 DO
```

is subject to this form of overhead in most compilers. (Some compilers provide for the case of constant step and termination and permit the programmer to specify that.)

Also, failure to use the call-by-name method of parameter passing can lead to unwanted overhead when arrays are being used between a pair of procedures. There is no way to specify whether subprograms are to be open or closed.

Thus, none of the requirements J1 through J5 are satisfied by ALGOL 60.

## 12. PROGRAM ENVIRONMENT

ALGOL 60 has been implemented in a number of environments. Although it does not require an operating system for its execution, it does require extensions for input/output to be usable in solving real programming problems. ALGOL 60 is a block structured language; there is no conceptual problem with having an inner block which is separately compiled. However, such a feature is not supported by the language.

Some ALGOL 60 environments, most notably those on Burroughs computers, provide a useful set of programming tools and aids, along with a set of extensions to the basic language which assist in monitoring variables and debugging. In general, the interface for this set of tools is an operating system so that one may save files and edit them or compile them or execute them as appropriate. Thus K3 and K4 are partially satisfied.

Much of the early work on verification and inductive assertions was based upon ALGOL 60. Assertions and other similar features may be introduced as comments. Some derivatives of ALGOL 60 incorporate assertions in a way that facilitates machine processing and checking.

13. TRANSLATORS

There are examples of ALGOL 60 translators which represent both subsets and supersets of the reference language. Burroughs' and numerous other implementations have extended the language to include input/output. Most implementations also have eliminated some of the features of the standard, including dynamic own arrays and numeric labels. Thus, both L1 and L2 are not met.

Because the translators developed for ALGOL 60 vary so greatly, it is not possible to give evaluations for ALGOL 60 on items L3-L8. It is clear that ALGOL 60 can be implemented on a number of machines. Experience has shown that the stack-oriented architecture exemplified by the Burroughs computers is particularly efficient for the implementation of ALGOL and its block structure.

While it would be possible to write an ALGOL 60 compiler in ALGOL 60, that would involve the use of integers to represent characters. Such a compiler would be particularly opaque and would be highly dependent upon the host machine.

14. LANGUAGE DEFINITION, STANDARDS, AND CONTROL

ALGOL 60 was composed of a number of features which were considered revolutionary at the time of their design. Some of them were not fully understood, a fact which resulted in some ambiguities in the original language and some trouble spots which have only been cleared up by the publication of the "New" ALGOL 60 report in 1976. However, all of the features of the language are now very well understood. There exist numerous tutorial books on the language. (C1 and C3 are met.)

ALGOL 60 was perhaps the first language to be defined unambiguously using a formal definition for the syntax. The semantics were specified in English, however. The BNF notation which is now commonly used for syntax description was first used in conjunction with the ALGOL development effort.

Maintenance of ALGOL 60 is vested in IFIP Working Group, 2.1 the ALGOL Subcommittee of IFIP TC2 on Programming. They publish a newsletter, the "ALGOL Bulletin," dealing with ALGOL 60 and ALGOL 68. They do not have the responsibility for language implementations; that lies with the individual vendors. There are no official libraries of ALGOL 60 programs. However, ALGOL 60 has long been used as the basis for the composition of algorithms intended for computer execution. Many of these algorithms have been published in collections of algorithms by ACM (originally in Communications of the ACM, now in Transactions on Mathematical Software) and in Numerische Mathematik. In general, these algorithms are not in machine-readable form and there are no standards for distribution of ALGOL 60 programs.

# SECTION 3. LANGUAGE EVALUATION SUMMARY FOR ALGOL 60

HOL EVALUATION PROJECT:
LANGUAGE EVALUATION SUMMARY FOR ALGOL 60

A. DATA AND TYPES

A01 S
A02 N
A03 N
A04 N
A05 N
A06 P
A07 N

B. OPERATIONS

B01 P
B02 N
B03 S
B04 P
B05 S
B06 P
B07 N
B08 P
B09 N
B10 N
B11 N

C. EXPRESSIONS AND PARAMETERS

C01 S
C02 P
C03 S
C04 N
C05 P
C06 S
C07 P
C08 N
C09 N

D. VARIABLES, LITERALS, AND CONSTANTS

D01 N
D02 P
D03 N
D04 N
D05 P
D06 N

E. DEFINITION FACILITIES

E01 N
E02 N
E03 S
E04 N
E05 N
E06 P
E07 S
E08 N

F. SCOPES AND LIBRARIES

F01 S
F02 N
F03 S
F04 U
F05 U
F06 U
F07 N

G. CONTROL STRUCTURES

G01 P
G02 N
G03 P
G04 P
G05 P
G06 N
G07 N
G08 N

H. SYNTAX AND COMMENT CONVENTIONS

H01 S
H02 S
H03 S
H04 P
H05 N
H06 P
H07 P
H08 S
H09 N
H10 S

I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

I01 S
I02 P
I03 N
I04 N
I05 S
I06 N
I07 S

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCE

J01 U
J02 S
J03 N
J04 N
J05 N

K. PROGRAM ENVIRONMENT

K01 S
K02 N
K03 U
K04 U
K05 N

L. TRANSLATORS

L01 N
L02 N
L03 U
L04 U
L05 U
L06 U
L07 U
L08 U
L09 P

M. LANGUAGE DEFINITION, STANDARDS AND CONTROL

M01 S
M02 P
M03 S
M04 N
M05 N
M06 P

Key: S = Satisfied
P = Partially Satisfied
N = Not Satisfied
U = Undetermined

# SECTION 4. DETAILED COMPARISON OF ALGOL 60 TO THE TINMAN

## 1. INTRODUCTION

This section outlines specific evaluations of each TINMAN requirement as they are (or are not) met by the language ALGOL 60.

Page number references when given refer to the basic reference document for the language: P. Naur (Editor), "The Revised Report on the Algorithmic Language ALGOL 60," Communications ACM, January 1963.

A. DATA AND TYPES

A1. Typed Language
Satisfied

A2. Built-in Data Types
Not Satisfied

No records.

A3. Precision Declaration
Not Satisfied

No precision specification.

A4. Fixed Point
Not Satisfied

Integers only; no fixed point decimals

A5. Enumeration Sense of Character Sets
Not Satisfied

A6. Arrays
Partially Satisfied

An array of variable size may be declared in a procedure or function, then allocated at execution time depending upon the value of dynamically determined variables. Furthermore, such arrays can be deallocated and reallocated with a different size during program execution. However, the number of dimensions and typing of arrays is fixed.

A7. Record Equivalence
Not Satisfied

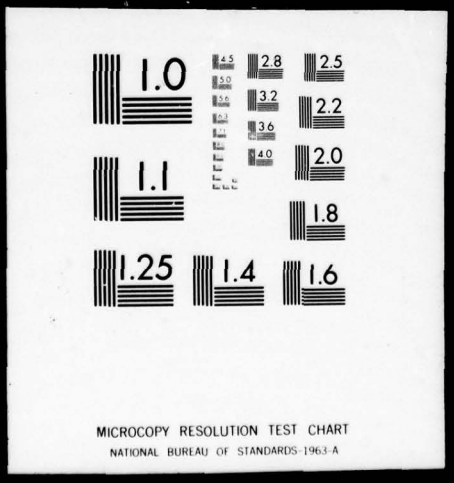
No records.

AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

6 OF 28
ADA037634

6 OF 28
DA037634



B. OPERATIONS

B1. Assignment
Partially Satisfied

B2. Equivalence Operator
Not Satisfied

ALGOL 60 requires = for numerics and ≡ for Booleans. Comparison between numerics and Booleans is an error.

B3. Relationals Defined
Satisfied

Note: there is no enumeration.

B4. Arithmetic Operations
Partially Satisfied

No fixed point. / accepts operands of real or integer; ÷ applies only to two integers.

B5. Truncation/Rounding
Satisfied

B6. Boolean Operations
Partially Satisfied

Provides ∧, ∨, >, no nor; does not short circuit.

B7. Scalar Operations
Not Satisfied

Must index elements

B8. Type Conversion
Partially Satisfied

No fixed point

B9. Range Exceptions
Not Satisfied

No exception handling for run-time truncation; however, run-time truncation should not occur. No numeric ranges.

B10. I/O Operations
Not Satisfied

No I/O facilities in ALGOL 60 - I/O defined in New ALGOL 60 Report.

B11. Power Set Operations
Not Satisfied

No power sets of enumeration provided.

## C. EXPRESSIONS AND PARAMETERS

C1. Side Effects
Satisfied

C2. Operand Hierarchy
Partially Satisfied

Nine levels of operator hierarchy

C3. Expressions Permitted
Satisfied

C4. Constant Expressions
Not Satisfied

No specification for constants

C5. Parameter Rules
Partially Satisfied

The declaration of an array within a procedure need not show dimensions; no exception handling, no type declarations; no parallel processing.

C6. Type Agreement
Satisfied

C7. Formal Parameter Kinds
Partially Satisfied

Call be value and call by name. Procedures as parameters are handled as call by name. No class for specifying control action when exceptions occur.

C8. Formal Parameter Specifications Optional
Not Satisfied

Type must be specified. Dimension and number of dimensions need not appear in formal parameter list. However, use of array within procedure yields number of dimensions.

C9. Variable Number of Parameters
Not Satisfied

One can pass arrays of variable size, but no argument lists of variable length.

D. VARIABLES, LITERALS AND CONSTANTS

D1. Constant Value Identifiers
Not Satisfied

D2. Numeric Literals
Partially Satisfied

No specifications for precision of real literals between object machines.

D3. Initialization
Not Satisfied

D4. Ranges and Stepsize
Not Satisfied

D5. Variable Types
Partially Satisfied

No records, but true within data types supported by language.

D6. Pointer Variables
Not Satisfied

No pointers

E. DEFINITION FACILITIES

E1. Type, Operator Definitions
Not Satisfied

E2. Consistent Use
Not Satisfied

No defined types

E3. No Default Declarations
Satisfied

E4. Extend Existing Operators
Not Satisfied

No new types

E5. Type Definitions
Not Satisfied

E6. Data Definitions
Partially Satisfied

Arrays only; no records, power sets, enumeration, discriminated union.

E7. No Free Union
Satisfied

E8. Type Initialization and Allocation
Not Satisfied

Not applicable

F. SCOPES AND LIBRARIES

F1. Allocation and Access
Satisfied

F2. Limit Access Scope
Not Satisfied

No type definitions

F3. Static Scope Determination
Satisfied

F4. Libraries Available
Undetermined

F5. Library Contents
Undetermined

Implementation dependent--not prohibited by Reference Language.

F6. Libraries and Compools
Undetermined

F7. Machine Dependent Interface
Not Satisfied

No I/O

G. CONTROL STRUCTURES

G1. Control Structures
Partially Satisfied

No parallelism, asynchronous interrupts, or exception-handling.

G2. GOTO
Not Satisfied

Can branch out of local scope.

G3. Conditional Control Structures
Partially Satisfied

Cannot branch on discriminated union; else not required with if...then.

G4. Iteration Control
Partially Satisfied

Termination at loop beginning or via branch out of loop; control variable need not be local.

G5. Recursive Routines
Partially Satisfied

Can define procedures within recursive procedure.

G6. Parallel Processing
Not Satisfied

G7. Exception Handling
Not Satisfied

No exception-handling

G8. Synchronization Primitives
Not Satisfied

Not specified in language; OS dependent

H. SYNTAX AND COMMENT CONVENTIONS

H1. Syntax Characteristics
Satisfied

H2. No Syntax Extensions
Satisfied

H3. Source Character Set
Satisfied

H4. Literals and Identifiers
Partially Satisfied

No break character for use internal to identifiers or literals.

H5. Continued Lexicals Units
Not Satisfied

No End-of-line in literal strings

H6. Keywords
Partially Satisfied

Implementation dependent - about 22 keywords + those added for I/O in some implementations.

H7. Comment Convention
Partially Satisfied

Requires explicit comment terminator.

H8. Unmatched Parentheses
Satisfied

H9. Uniform Referent
Not Satisfied

[ ] for subscripts; ( ) for function calls

H10. Consistency of Meaning
Satisfied

I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

I1. Defaults in Program Logic
Satisfied

I2. Default Override
Partially Satisfied

Programmer cannot override defaults.

I3. Compile-time Parameters
Not Satisfied

No compile-time facilities

I4. Object Environment Compile-time Parameter
Not Satisfied

I5. Language Kernel
Satisfied

I6. Translator Restrictions Defined
Not Satisfied

No translator limits defined

I7. Object Environment Restrictions
Satisfied

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCE

J1. Efficient Code
Undetermined

Dynamic own arrays are treated differently.

J2. Optimization Invisible
Satisfied

J3. Access to Machine Language
Not Satisfied

ALGOL 60 does not permit machine code insertions - not a systems programming language.

J4. Implementation Representation Tradeoffs
Not Satisfied

J5. Open/Closed Implementations
Not Satisfied

# CHAPTER 5
# ALGOL 68

The contents of this chapter of the report involve material relating to ALGOL 68.

Section 1. Evaluation of ALGOL 68 Against Language Design Criteria

Section 2. Evaluation of ALGOL 68 Against the TINMAN

Section 3. Language Evaluation Summary for ALGOL 68

Section 4. Detailed Comparison of ALGOL 68 to the TINMAN

# SECTION 1. EVALUATION OF ALGOL 68 AGAINST LANGUAGE DESIGN CRITERIA

## 1. INTRODUCTION

This section examines the extent to which ALGOL 68 satisfies the criteria for a "good" higher order language as presented in "Criteria for Evaluation of High-Order Languages." ALGOL 68 is seen to fulfill the requirements quite well. The primary areas of weakness of ALGOL-68 are the absence of abstract data types and the size of the language.

## 2. ABSTRACTION

ALGOL 68 provides excellent facilities for procedural abstraction. It is possible to declare a proc which accepts two types of parameter: those called by value and those called by ref. The value returned by a proc may be an object of any mode, whether built-in or user-defined.

The facilities for operator and operand abstraction are also quite good, although ALGOL 68 does not support the concept of encapsulating the set of operations on a data type with the representation of the type (abstract data type). First, it is possible to define new operators as infix symbols in much the same way as one defines procedures, and to associate a priority with these operators. Thus, one could extend ALGOL 68 for sets, for example, and include a set of dyadic operators (or extend existing operators) to carry out the basic set of operations of union, intersection, complementation, subset testing, etc.

One can define new data types in a limited way. Record classes are provided (struct) with the generality that structures may be declared within structures may be declared within structures. Existing operators are extended to apply to substructures as appropriate and there are standard initialization facilities for objects of mode struct.

It would be relatively straightforward to extend ALGOL 68 to provide an abstract data type facility comparable to that of

Euclid (for example). What would be required would be an exports clause to identify the visible names, a rep clause which declared the representation of the object, and an ops clause to encapsulate the associated operators, thereby providing the extended mode in an abstract format.

3. MODULARITY

ALGOL 68 is a block structured language in which variables may be passed as parameters (reference or value) or may be shared through global declarations. There is no obvious mechanism whereby items declared in one block may be kept private from those blocks nested within it, as with the PRIVATE declaration of LIS.

While it is possible to achieve modularity in ALGOL 68 programs through the use of parameter passing, there are also a number of mechanisms for subverting modularity, caused primarily by the freedom to access global variables and by the capability to produce side effects in a value-returning proc. For example, the proc call f(x) may return a value for f, but could also change the value of x. In general, though a proc may represent a procedural module.

As with other block structured languages (ALGOL 60, PL/I) and quasi-block structured languages (PASCAL), modularity can be achieved, but only with some care on the part of the programmers. As with the other languages, there is no method for establishing "data modules."

4. LINEAR FLOW OF CONTROL

The control structures of ALGOL 68 include the go to statement, with no serious restrictions on its use. A go to can designate any label within the present scope of a program or within any enclosing scope. It is expected that the need for the goto will be quite limited, however, since ALGOL 68 supports a number of other control structures, including an if-then-else, various forms of a do statement, a case statement, and a while statement.

In general, the control structures for ALGOL 68 will not require the use of the go to with any frequency.

## 5. CONTROL OF SCOPE AND BINDING

Scope and binding of variables is properly managed in ALGOL 68, with variables declared upon entry to a block. Space is allocated upon block entry and deallocated upon exit, unless the object is declared with a heap generator, in which case it remains in existence until the program is terminated.

Variables are bound to a mode upon their creation and cannot change that mode. It should be noted that the mode may be a union of basic modes in the language, e.g., mode rent = union (real, int).

The scope of each declaration is unambiguous and visible from the program's static listing.

## 6. READABILITY AND COMPREHENSIBILITY

ALGOL 68 programs are generally comprehensible. Two language features which tend to detract from this readability should be cited, however. First, the symbols begin and end may be replaced by matched parentheses, and the construct if A then B else C may be replaced by (A|B|C). Second, some operations, such as plusab (equivalently :=+), conceal the fact that an assignment is made to a variable.

On the other hand, ALGOL 68 is exceptionally consistent as a language; an ALGOL 68 programmer quickly becomes familiar with major constructs and their use. Accordingly, it becomes relatively easy to understand ALGOL 68 programs. The most significant difficulty is caused by the size of the language and by the ability to define new operators. If a programmer has made liberal use of the permissible abbreviations for keywords and has defined a significant number of dyadic operands and/or changed the priority of existing operands, the structure of an ALGOL 68 program can become rather difficult to comprehend.

It seems that ALGOL 68 programs are somewhat less readable than programs in PASCAL, but rather more comprehensible than a comparable program in PL/I or FORTRAN.

## 7. PROCEDURES AND DATA

ALGOL 68 maintains a distinction between procedures and data. It is not possible to modify ALGOL 68 instructions or to read data and then execute it, as is the case in COBOL, MUMPS, SNOBOL4, and several other languages. Since variables must be declared, ALGOL 68 always makes it clear as to what is program text and what is data.

## 8. LANGUAGE SIZE

ALGOL 68 is a moderately large language in terms of the number of syntactic units required to represent the grammar. However, the number of keywords in the language is relatively small, well under 100. ALGOL 68 is defined with a W-grammar, which includes both syntactic and semantic elements. The beginner would have significantly more difficulty comprehending the ALGOL 68 report than other reports written in BNF or a less formal notation.

The size of ALGOL 68 is somewhat deceiving, because only a relatively small number of individual concepts are included. An excellent and largely complete tutorial booklet on the language is shorter than 200 pages, even shorter than the more formal reference report.

More substantial than the language size is ALGOL 68's own terminology, which will be particularly foreboding to a person accustomed to programming in other languages. After one gets past the initial barriers to comprehension of the language, the language is surprisingly manageable intellectually. The language is exceptionally smooth, with only a few inconsistencies or special requirements, e.g., casting. However, ALGOL 68's size presents substantial problems for the implementor, who cannot easily use the syntactic processing techniques that have been used for languages defined in BNF.

## 9. ANALYZABILITY

ALGOL 68 programs can rate fairly high on analyzability if they are written using the powerful control constructs above. The language contains the concept of "pragmats," denoted by pr, which are treated as comments. However, pragmats may contain assertions and could be used to generate verification conditions for program provers.

Similarly, test data can be generated and execution can be monitored in a relatively straightforward way in ALGOL 68. Thus, ALGOL 68 imposes few barriers to automated analysis.

## 10. SUMMARY

Although ALGOL 68 has some shortcomings when measured against this set of criteria, it also possesses the expressiveness and conceptual integrity which are at the heart of programming language design. Because the language features are largely orthogonal, it would be a straightforward exercise to delete certain features and propose others without drastically altering other language features. Thus, one immediately sees how to add abstract data types, sets, fixed point numbers, or other data types, while reducing the generality of the go to and eliminating such things a flex arrays and certain operators which make the language less satisfactory. As a result, it is extremely tempting to begin with the notions of ALGOL 68 as the basis for meeting the criteria of TINMAN, rather than making minor changes in a language which was constructed in an ad hoc manner to conform to TINMAN.

## SECTION 2. EVALUATION OF ALGOL 68 AGAINST THE TINMAN

## 1. OVERVIEW

ALGOL 68 provides a number of the facilities which are important in the creation of a programming language supporting a systematic approach to software development. With respect to the criteria of the TINMAN, ALGOL 68's greatest strengths lie in full declarations of data objects, the block structure of the language, the ability to extend binary operators to newly defined data types, support for virtually all of the basic data types, and a sharply defined concept of data type (termed mode). Its major weaknesses are the lack of abstract data types, the failure to support the uniform referent concept, and the nondeterministic evaluation order of expressions in ALGOL 68 collateral clauses. Going beyond the criteria of TINMAN, one can also criticize ALGOL 68 for some of the features which present implementation difficulties and conceptual difficulties for someone learning the language. In particular, the ALGOL 68 terminology and the presentation of the reference document are quite formidable, even for those with substantial exposure to programming languages.

Notwithstanding these points, ALGOL 68 must be considered a strong candidate (as the basis) for a DoD standard, with appropriate extensions for abstract data types and restriction on the use of collateral clauses. This acceptance would clearly have to be supported by extensive tutorial documentation and training. The following sections give an overview of the compatibility between ALGOL 68 and TINMAN, followed by a detailed comparison between the two. Emphasis is given to those aspects of the language which are particularly well handled by ALGOL 68 and to those which are serious detriments to the language with respect to TINMAN.

## 2. DATA AND TYPES

ALGOL 68 is a strongly typed language in which there are few unexpected conversions from one type to another. The six types of conversions are termed dereferencing, widening, rowing, deproceduring, voiding, and uniting. Dereferencing is done when an assignment statement of the form:

```
a := b
```

is performed, where a and b are declared to be of the same mode, real for example. b has mode ref real, as does a. In order to assign the value of b to a, it is necessary to obtain the value of b, thereby dereferencing b and producing a value of mode real.

Widening would occur if we had declared

```
real a, int b
```

with a := b, b would be of mode ref int. The integer value would have to be coerced to a real value to complete the assignment; this is termed widening. The reverse operation, had we declared

```
int a, real b
```

and written a := b, would be illegal, since dereferencing and widening would not apply. It would be necessary to use a monadic operator such as round or entier to make a proper assignment, e.g.

```
a := round b
```

Uniting permits an object to belong to a type union, e.g.

```
mode m = union (int, real)
```

Had we declared

```
m a,b
```

using this new mode, then the assignment a := b would work

if b were either int or real; a would then assume the corresponding mode.

All of these coercions among types are compatible with Tinman requirement A1, and also with B8 (except for the nonexistent fixed point types).

ALGOL 68 supports built-in modes of int, real, char, string, Boolean, bits, and bytes along with arrays of these modes and structs which provide a record-handling facility. The int and real specification may be preceded by one or more instances of the words short or long to impart information about the size of the object or the required precision. The interpretation of these words is strictly up to the implementor, though (and could be ignored). The ability to specify precision is not comparable to that facility as it exists in PL/1. (Precision specifications in PL/1 are a frequent cause of programming errors and unexpected results as a result of truncations of leading digits and roundoff errors.)

ALGOL 68 does not explicitly support fixed point arithmetic (as does COBOL, for example). However, one could devise a set of procedures and extend the available (and meaningful) binary operators as part of the standard prelude to add this facility. (Fixed point arithmetic is also a common source of errors in numerical computation.)

While ALGOL 68 requires declarations of arrays, with an explicit number of dimensions and bounds, it is also possible to declare arrays with the property flex which permits the bounds of those arrays to be changed dynamically. Flex arrays clearly violate the intent of TINMAN; they could be deleted from the language without major impact on other language features.

3. OPERATIONS

The ALGOL 68 language compares quite favorably with the TINMAN specifications regarding operations. Assignment

works in the desired manner (B1), relational operators are properly defined (B3), the basic built-in arithmetic operations are present for real and int values (B4), truncation and rounding will operate properly (although the implementation of the language may cause roundoff errors as a result of performing decimal arithmetic with a binary floating-point device (B5)), and all of the desired Boolean operations other than "nor" are present (B6). ALGOL 68, like PL/1, permits assignments of matching array rows or records without explicit indexing. Thus, the declaration

```
[-10:10] int a,b;
```

permits us to write

```
a := b.
```

Short circuit mode is not defined in the language, although at least one implementation of ALGOL 68 gives the user the ability to follow that alternative. The language design decision was to permit side effects associated with the evaluation of a Boolean expression, i.e., values changed as the result of assignments, function calls, or procedures.

ALGOL 68 has an extremely powerful facility for defining new binary operators and for extending binary operators to new types. While the concept of equality is defined for the built-in types, it is not automatically defined for new modes. However, if we introduce a new mode foo, we may then write

```
op = = (foo a,b)  bool;
```

and proceed to define the concept of equality for objects of mode foo, thereby making it possible to compare objects of type foo for identity. This notion can be extended to compare any two data objects regardless of type (B2). The number of definitions that would have to be made can become combinatorially large, since the operator must be explicitly defined for every conceivable pair of types. Thus for all practical

purposes, ALGOL 68 does not meet the Tinman requirement. A more realistic requirement would be that the equality test exists between any two objects of the same mode; this is in conformance with the equality test present in the language CLU and is quite workable. The intelligent programmer should not be in the position of wanting to compare objects of different mode as a general rule: int/real and char/string are typically the only meaningful mixed mode comparisons.

ALGOL 68 provides operations beyond those specified by Tinman. It is often the case that one will write

```
I := I + 1
```

in a program, where I is of mode int. ALGOL 68 provides the abbreviated operator +:= (or plusab) permitting this to be written as I plusab 1. Similar operators exist for all of the fundamental arithmetic operations so that one could write

```
p timesab q + 5
```

which means

```
p := p *(q+5).
```

The assignment operation is hidden by the operator so that the reader of the program does not immediately observe the assignment. The specification of these operators requires that the left-hand-operator be of type ref int or ref real so that

```
5 + q plusab p
```

is illegal since 5+q does not coerce to mode ref real or ref int. The operators plusab, minusab, timesab, divab, overab, and modab should be removed from the language in order to conform to the specifications of Tinman.

ALGOL 68 frequently permits more than one symbol to stand for an operation, e.g. +:= and plusab or (worse) mod, ÷*, ÷x, %*, and %x. While this is sometimes necessitated

by constraints on the size of the character set, a more workable approach would define the language in terms of the ASCII character set and provide a single symbol for each. The interchangeability of matched parentheses with begin and end, although justified by the places in which a closed clause may appear, can be particularly annoying. Programming standards would have to be established in order to produce readable programs and identifiable blocks in ALGOL 68 programs.

4. EXPRESSIONS AND PARAMETERS

ALGOL 68 has a serious conflict with Tinman with respect to C1. The order of evaluation for arguments of an expression is not specified in the ALGOL 68 Report. In fact, it is deliberately left unspecified, in order to give freedom to the implementor to make certain optimizations. If one wrote

```
S := F(X) + G(X)
```

there would be no guarantee of the order of evaluation of F and G. Should either of the functions change the value of X (a side effect), the value of the function would be different (potentially) for the two orderings of evaluation. Collateral elaboration is a fundamental concept in ALGOL 68, and one must go to considerable care in programming to avoid the effects of collateral elaboration rather than serial elaboration. A fairly strict programming discipline regarding function calls and the implications of side effects would be required, since imposition of a left-to-right ordering would fundamentally change the language, necessitating significant changes in the language definition.

ALGOL 68 has more levels of operator hierarchy than are generally considered to be desirable. (In contrast with LIS, which has 4, ALGOL 68 has 10 levels, with the plusab

(etc.) operators at level 1 and some of the monadic operators like upb and lwb at level 10.) Significant also is the fact that the ALGOL 68 programmer can change the priority of any operator through the prio declaration, in addition to specifying the priority of newly defined binary operators. Thus, the standard prelude of the language could be modified to define a smaller number of levels of operator hierarchy. However, the large number of levels of hierarchy are a result of the concept of collateral elaboration. If all operators were at the same level of hierarchy, the expression

```
a + b x c ** d
```

could very well evaluate differently under different implementations of the language. (This is a tradeoff factor in language design.)

ALGOL 68 meets the other requirements with respect to expressions and parameters quite well. Expressions are permitted wherever both constants and references to variables of that type are allowed (C3), although there are instances of potential ambiguity which must be resolved by casting. Consider

```
begin int i := 0, k := 1;
ref int ptr := i;
     ptr:= k;
     print (i);
end;
```

The assignment ptr := k can cause trouble, since ptr is of mode ref ref int and k is of mode ref int. The standard dereferencing must not occur here, since ptr points to i, and the assignment would cause i to be assigned the value 1, which is quite different from the evident intent. Thus, the assignment statement must be replaced by

```
ref int (ptr) := k;
```

casting the type of the result.

Rules for parameters are uniform: ALGOL 68 supports calls by reference and by value in a manner corresponding closely to PASCAL. (ALGOL 68's call by reference is different from the ALGOL 60 call-by-name concept.) Operands to which assignments can be made carry with them the symbol ref to designate call-by-reference. Hence

```
proc add = (int a, int b, ref int c) void;
     begin c := a + b end
```

is legal. Note that if c was of mode int rather than mode ref int, the assignment statement would not be permitted. This is a problem in FORTRAN which does not make the proper differentiation; one could write

```
SUBROUTINE ADD (A,B,C)
C = A+B
RETURN
END
```

and legally write CALL ADD (2,3,4), which is not permitted in ALGOL 68, in which the type of the actual parameter is compared with that of the formal parameter. In this instance, 4 would not be coerceable to mode ref int and an error would be signalled.

ALGOL 68 permits procedure names to be passed in the same manner as other parameters and even returned as results. They do not constitute a separate class of parameters in the sense of TINMAN's C7. Also, ALGOL 68 has no provision for exception-handling or for passing exception conditions as parameters. Extension of the language for exception-handling is quite complicated in order to control variables and their scoping properly.

Variable length parameter lists cannot be passed in ALGOL 68. However a pointer can point to a linked list of objects so that one can pass the pointer as a parameter, giving the effect of passing a variable number of items when only passing one.

Types of formal parameters must also be specified (in conflict with C8), as must the number of dimensions for arrays. The concept of generic procedures may be achieved by making use of type unions. A procedure designed to handle objects of mode int, real, or bool can have

```
mode mess = union (int, real, bool)
```

and also the declaration

```
proc p = ( ... mess m, ref mess q, ... ) ...
```

in a parameter list to permit treating parameters of differing type. If a new mode is added, it is only necessary to alter the mode declaration, and make any necessary discrimination based on the mode within the procedure p. The ALGOL 68 solution appears somewhat more elegant than the TINMAN requirement C8 will permit.

## 5. VARIABLES, LITERALS, AND CONSTANTS

ALGOL 68 provides for the definition of constants and the initialization of variables. There is only a slight difference in syntax, which could be the cause of some programming errors. For example, we can write

```
int i = 2, j := 3;
```

which declares i to be a constant of value 2 and j to be an int variable with initial value 3.

ALGOL 68 does not provide a facility to limit the range of numeric variables along the lines of the PASCAL subrange mechanism. Such a facility would lead to some confusion about the concept of a mode. It is clear the TINMAN's specification E7 does not require a subrange to be treated as a new mode (as is done in CS-4).

ALGOL 68 provides a pointer mechanism which is safe in its use. One does not declare an object to be of mode ref, but rather of mode ref mode, where mode is the mode of

object to which the variable can point. This cannot be changed within the program. ALGOL 68 permits multiple level pointers as does PASCAL. A variable may be declared to be of mode ref ref ref ref int, for example.

## 6. DEFINITION FACILITIES

The user of ALGOL 68 has a limited facility for the definition of new modes. One may introduce modes which include structures, unions of types, structures dependent upon other variables (corresponding to the variant case RECORD structure of PASCAL but not as general as plexes in LIS). One may also define operations on those new modes, either by defining procedures or functions which accept objects of those modes as parameters, by defining new dyadic (binary) or monadic (unary) operators, or by extending existing operators to apply to the new modes. The set of built-in modes is sufficiently extensive to provide a good facility for introducing new modes. The only mode which is absent is that of sets with the associated powerset mechanism. If a set mechanism was present in the language, there is some chance that the designers would have eliminated pointers from the language, since it would then be possible to define recursive data structures without pointers (as described by Hoare).

ALGOL 68 is particularly attractive in the manner in which existing operations in the language may be used to make defined modes indistinguishable from built-in modes. The concept of operator extension to new modes is not present in many languages. New modes may be initialized (E8), but cannot be automatically finalized.

ALGOL 68 supports the notion of abstract data types as specified by E5. There is no encapsulation of the operators on a given mode with the definition of the representation of the mode. Hence, the representation of the mode remains visible. The necessary facilities for mode definition and

operator extension are present, and it would be feasible to extend the language to support such a capability. A careful programmer could achieve much of the same effect as with abstract data types even in their absence; one does not have the option, however, of incorporating an abstract type (such as a stack) in hardware or as a separately compiled module.

Type union capabilities are present with the ability to declare that an object may be one of a number of modes. Mode definition is by discriminated union rather than by free union; i.e., it is possible to determine the mode of the object through a case statement at any point.

The definition facilities of ALGOL 68 are quite powerful, but fail to provide for power sets of enumeration modes or for abstract types.

## 7. SCOPES AND LIBRARIES

ALGOL 68 has mechanisms for distinguishing between the scope of allocation and the scope of access. Scope of access rules are similar to ALGOL 60, with a block structure in which names may be redefined within inner blocks. Control over allocation differs from that of ALGOL 60. A trouble spot in ALGOL 60 was own variables: their initialization versus dynamic allocation. These concepts have been replaced by loc and heap allocation in ALGOL 68. Heap encompasses the concept of own by providing for continuing existence of an object with its value, but preserving limited access to the object. It is not possible to limit the access to separately defined structures other than through this mechanism; i.e., no abstract types limit access to a fixed set of operators.

ALGOL 68 does not make lengthy statements about the run-time environment for a program. There are a number of built-in functions and procedures, and it should be a straightforward task to provide libraries of other useful functions and procedures. As with most high level programming langauges, the translator would exist on a computer system which supports

a library of programs on secondary storage which can be directly executed. The base language does not provide straight text insertion (F5) as does RATFOR, for example. Many of the issues implicit in F5 and F6 are dependent upon the implementation of the language rather than upon its definition.

ALGOL 68 has machine-independent procedures for input/output. However, there is no built-in mechanism for associating a logical file with a physical file, which must be done through a job control mechanism. There is no inherent language feature to prevent interface to machine dependent hardware capabilities.

8. CONTROL STRUCTURES

ALGOL 68 supports a variety of control structures, including the go to, bracketed if-then-else and case statements, a while statement and a general looping capability which incorporates controlled iteration (the for statement). When a for is used, the control variable is not accessible outside the loop it controls; the control variable is treated as a strictly local declaration within the loop, and is given a constant value on each iteration of the loop.

ALGOL 68 also supports concurrency (pseudo-parallelism) through the down and up facilities, which provide the concept of semaphores, called sema mode in ALGOL 68. The par declaration prior to a set of closed clauses specifies they are to be treated as parallel in their execution.

The facilities of ALGOL 68 are very primitive with respect to asynchronous interrupt handling and exception handling. Except for input/output conditions, there is no exception handling for faults such as division by zero, overflow, etc. There is no way for users to easily define or signal exceptions.

The bracketed if-then-else and case statements provide full partitioning for choices and avoid the ambiguity of the dangling else (if A then if B then C else D). They may also be used to discriminate the type of an object defined as having a united mode.

The conditions of G4 are met except for loop termination, which must occur either at the end of the loop or via an unconditional branch out of the loop. There is no correspondence to the leave feature of BLISS.

Recursion is provided in its full generality.

In summary, the control structures of ALGOL 68 are among the most powerful available in any programming language, possibly excepting PL/I. The flaws in PL/I control structures lie in the looping (which ALGOL 68 treats more effectively) and in the unconditional branching permitted with ON-conditions.

9. SYNTAX AND COMMENT CONVENTIONS

ALGOL 68 permits a well structured and highly readable program to be created with appropriate indentation of lines (though not done automatically as in LIS), mnemonic variable names, and ample comments. It is also possible to write less readable ALGOL 68 programs by ignoring indentation, by choosing meaningless variable names, etc.

ALGOL 68 has a well-defined formal grammar, called a W-grammar or two-level grammar. Such a grammar is not parsed as easily as BNF, and presents some minor implementation problems; the W-grammar does provide additional language power. Early versions of ALGOL 68 were context-sensitive, but the Revised ALGOL 68 Report is syntactically context free.

Some of the keywords in ALGOL 68 are themselves abbreviations (e.g., int, bool, and sema), but abbreviations

of keywords are not possible in the PL/I sense of DCL and DECLARE.

An ALGOL 68 programmer has the power to modify the source language syntax by changing the priority of operators. The ability to define priorities is essential for creating new dyadic operators for defined modes. There is no easy way to prevent this operation from being applied to existing built-in modes. Limiting this capability thus detracts from the usability of ALGOL 68's language extension facilities.

The language defines formation rules for identifiers and literals. The blank character " " may be used as a break character internal to identifiers. Character literal strings may be separated by quote marks ", so that "abc""def" is equivalent to "abcdef". Spacing may be used freely to improve readability; only basic symbols cannot have internal spaces.

ALGOL 68 is a fairly large language in terms of the complete number of constructs which are permitted, and the different modes which are supported. This fact is reflected in the number of keywords in the language (approximately 50 in contrast to only 22 in ALGOL 60). This number is quite small when compared with COBOL, which has more than 300 reserved words. Keywords are generally meaningful, although many objections have been raised to use fi, od, and esac as terminators of if, do, and case.

ALGOL 68 has four commenting conventions, permitting use of the word comment, co, #, and ¢, in matched pairs to begin and terminate comments. The availability of multiple terminators makes use of specific characters within the comment easier. There is no automatic end-of-line terminator for comments, however, since ALGOL 68 is a free-format language. Input is treated as a continuous stream; the end-of-line requirement seems to contradict TINMAN requirement H1.

While ALGOL 68 has uniformity with respect to matching of parentheses and well-defined meanings of language symbols, it does not support the uniform referent concept. For example, arrays are designated with brackets [], while function calls use parentheses ().

## 10. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

ALGOL 68 has no defaults in programs, unlike the situation with PL/I for example. Objects must be declared to have a mode and the language processor assigns storage on the basis of that mode declaration. The only way in which these defaults can be overridden is to declare new modes through the struct capability using the built-in modes bytes and bits to define the structure. It is generally not practical to override these defaults. In fact, to support the notion of abstraction properly the programmer should not have the opportunity to override the default object representation.

ALGOL 68 has no facilities for conditional compilation. The macro facility in ALGOL 68 is non-existent, as one cannot even declare a subroutine to be inserted as an open subroutine rather than as a closed subroutine.

ALGOL 68 also fails to set explicit limits on such items as the number of array dimensions, the length of identifiers, or other implementation issues, leaving each implementor free to make those decisions, possibly at the loss of portability between programs. MUMPS, for example, has a set of limits (termed a Level One Standard) that establishes portability conventions.

It is arguable whether ALGOL 68 contains a "simple clearly identifiable base or kernel which houses all the power of the language." ALGOL 68 is an extremely powerful language, but there is considerable disagreement as to its simplicity. ALGOL 68 features are largely orthogonal, i.e. each language feature contributes a single unique capability.

## 11. EFFICIENT OBJECT REPRESENTATION AND MACHINE DEPENDENCIES

The ALGOL 68 language report makes no comments with respect to the issues raised in this section of the TINMAN. The source language requirements indicated by J3, J4, and J5 are not present. The TINMAN requirement about optimizations and the effect of the program is handled by the notion of collateral elaboration. The language makes no statement regarding the order of expression evaluation to assure a program remains "correct" regardless of any optimizations made. An ALGOL 68 programmer must be aware of this fact and not write code which depends upon a particular order of expression evaluations.

## 12. PROGRAM ENVIRONMENT

ALGOL 68 makes few assumptions about the implementation of the language. While it is certainly desirable to incorporate ALGOL 68 into a programming environment with a powerful complement of tools, this issue is separate from the language definition issues. Furthermore, the availability of subroutine libraries, programming tools, etc., imply the existence of sophisticated operating system facilities, in contradiction to the assumption of K1.

ALGOL 68 permits the inclusion of assertions, compiler directives, etc. as comments. Rather than treating them as comments, ALGOL 68 introduces the concept of a pragmat, which is treated as a comment by the compiler but which might provide information for verification programs, etc.

## 13. TRANSLATORS

TINMAN states that all translators for the language must implement the complete language, with no subsets or supersets. ALGOL 68 makes provision for these cases. Furthermore, many of the existing implementations of ALGOL 68 are subset implementations, a fact resulting primarily from the

complexity of the language.

The TINMAN requirement that the translator minimize compiler costs appears to contradict the original language goal of efficiency. If most of the computing effort is to be in program execution (possibly in a production setting), then the critical factor is program execution time rather than program compilation time. The efficiency criterion established appears to indicate that such was the case. In that setting, then, extra compilation time could be used to optimize the object code produced, hence leading to more efficient programs.

The ALGOL 68 language provides adequate facilities for writing the compiler in ALGOL 68. The structure of translators and the certification of those translators with respect to diagnostic messages and compile-time checking is more of an implementation issue and an enforcement issue than a language design issue. These requirements should be imposed once a language has been selected to make sure that the translators behave similarly and that programs written for one environment will run in another. No such issues are explicitly treated in the ALGOL 68 report.

14. LANGUAGE STANDARDS, DEFINITION, AND CONTROL

This TINMAN section addresses itself primarily to the construction of a new language rather than to the evaluation of existing languages. It is thus hard to compare ALGOL 68 against these requirements.

ALGOL 68 must be viewed as being highly innovative in its use of language features, terminology, and related topics. It was designed with great care and the language is quite smooth considering the extent of the features.

The syntax and semantics of ALGOL 68 are defined unambiguously in the Revised Report on ALGOL 68. In fact, the

definition is such that many of the semantic notions are incorporated into the grammar. The grammar was developed by van Wijngaarden specifically for this language. Grammars of this two-level type are now called W-grammars.

User documentation has recently become available. Pagan's A Practical Guide to ALGOL 68 is keyed to the revised report and is extremely readable. A new edition of Lindsey and van der Meulen's Informal Introduction to ALGOL 68, which is somewhat more formal, is in press. An excellent ALGOL 68 tutorial by Tanenbaum appears in the June, 1976 issue of Computing Surveys.

# SECTION 3. LANGUAGE EVALUATION SUMMARY FOR ALGOL 68

HOL EVALUATION PROJECT:
LANGUAGE EVALUATION SUMMARY FOR ALGOL 68

A. DATA AND TYPES

A01 S
A02 P
A03 P
A04 N
A05 P
A06 P
A07 P

B. OPERATIONS

B01 S
B02 P
B03 S
B04 P
B05 S
B06 P
B07 S
B08 P
B09 N
B10 S
B11 N

C. EXPRESSIONS AND PARAMETERS

C01 N
C02 P
C03 S
C04 N
C05 P
C06 P
C07 P
C08 N
C09 P

D. VARIABLES, LITERALS, AND CONSTANTS

D01 S
D02 P
D03 S
D04 N
D05 S
D06 S

E. DEFINITION FACILITIES

E01 S
E02 S
E03 S
E04 S
E05 N
E06 P
E07 S
E08 N

F. SCOPES AND LIBRARIES

F01 S
F02 N
F03 S
F04 U
F05 U
F06 U
F07 S

G. CONTROL STRUCTURES

G01 P
G02 N
G03 P
G04 P
G05 P
G06 S
G07 P
G08 N

H. SYNTAX AND COMMENT CONVENTIONS

H01 P
H02 P
H03 S
H04 S
H05 P
H06 P
H07 P
H08 S
H09 N
H10 S

I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

I01 S
I02 P
I03 P
I04 N
I05 P
I06 N
I07 S

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCE

J01 U
J02 N
J03 N
J04 N
J05 N

K. PROGRAM ENVIRONMENT

K01 S
K02 N
K03 U
K04 U
K05 S

L. TRANSLATORS

L01 N
L02 N
L03 U
L04 S
L05 U
L06 U
L07 N
L08 S
L09 S

M. LANGUAGE DEFINITION, STANDARDS, AND CONTROL

M01 S
M02 S
M03 S
M04 N
M05 N
M06 N

Key: S = Satisfied
P = Partially Satisfied
N = Not Satisfied
U = Undetermined

# SECTION 4. DETAILED COMPARISON OF ALGOL 68 TO THE TINMAN

## 1. INTRODUCTION

This section outlines specific evaluations of each TINMAN requirement as they are (or are not) met by the language ALGOL 68.

Page number references when given refer to the basic reference document for the language: A. Van Wijngaarden, et.al., "Revised Report on the Algorithmic Language ALGOL 68," Acta Informatica, Volume 5, pages 1-236, 1975.

A. DATA AND TYPES

A1. Typed Language
Satisfied

It is possible to pass variable length arrays of a declared mode as parameters. Variables may be specified to be the union of two or more types: union (int,char) abc.

A2. Built-in Data Types
Partially Satisfied

No fixed point

A3. Precision Declaration
Partially Satisfied

Variables can be declared real, long real, long long real, short real, etc. to provide varying precision. The meaning of the modifiers is implementation-dependent. Some implementations of ALGOL 68 do not support this feature.

A4. Fixed Point
Not Satisfied

A new mode with operators can be defined to accomplish this.

A5. Enumeration Sense of Character Sets
Partially Satisfied

A mode can be defined to have the desired elements and desired ordering:

ASCII = [1:128] char = (skip, skip..."A","B",...)

A6. Arrays
Partially Satisfied

An array of variable size may be declared in a closed clause or a procedure. One can declare:

flex [n:n,p:1] int aa

or flex [1:0] real c

The dimensions of c become [1:3] upon execution of:

c:= (1.5,2.5,3.5)

The number of dimensions and the type of values remains static.

A7. Record Equivalence
Partially Satisfied

Single struct only, but flexible arrays and proc's may be part of a struct. Union Type may be used to achieve this objective.

B. OPERATIONS

B1. Assignment
Satisfied

B2. Equivalence Operator
Partially Satisfied

Standard prelude defines the modes which can be compared for equality, but the operator = (or eq) can be extended to compare objects of any mode, should new modes be introduced.

B3. Relationals Defined
Satisfied

B4. Arithmetic Operations
Partially Satisfied

÷ returns an integer value upon dividing two integers

/ returns a real permitting division of two reals, or of an integer by an integer, or of one by the other. (ref. 10.2.3.3b, 10.2.3.4, 10.2.3.5.)

B5. Truncation/Rounding
Satisfied

This is dependent upon the implementation's handling of fixed point overflow. The absence of fixed point facilities helps in meeting this condition.

B6. Boolean Operations
Partially Satisfied

Nor = >(a B) not built-in, but can be easily defined as:

op ≮ nor ≯= (bool a,b) bool : (a b/false/true)

No short-circuit; evaluations carried out (IBM 360 ALGOL 68C has ANDF, ORF).

B7. Scalar Operation
Satisfied

Also permits slicing of arrays

B8. Type Conversion
Partially Satisfied

Widening is a form of coercion - conversion from int to real, bits to bool, and bytes to [ ]char are done automatically when needed. No fixed point.

B9. Range Exceptions
Not Satisfied

No exception-handling for run-time truncation; however, run-time truncation should not occur. No range in ALGOL-68.

B10. I/O Operations
Satisfied

Control information - new page, new line, specific character position, backspace. Exceptions - end logical file, end physical file, value error, char error.

B11. Power Set Operators
Not Satisfied

No power sets of enumeration provided.

## C. EXPRESSIONS AND PARAMETERS

C1. Side Effects
Not Satisfied

ALGOL 68 has collateral clauses in which expression order is expressly undefined. (Ref. 2.1.4.2e)

C2. Operand Hierarchy
Partially Satisfied

There are 10 levels of operator hierarchy, which may be changed by the programmer.

C3. Expressions Permitted
Satisfied

C4. Constant Expressions
Not Satisfied

No specification concerning compile-time evaluation of constant expressions. An implementation may do this.

C5. Parameter Rules
Partially Satisfied

Procedures may receive arrays where the size is unknown.

C6. Parameter Agreement
Partially Satisfied

Limited forms of widening, e.g. int to real are permitted.

C7. Formal Parameter Kinds
Partially Satisfied

No class for specifying control action when exceptions occur.

C8. Formal Parameter Specifications Optional
Not Satisfied

Note conflict with C6. Dimension is variable, but number of dimensions isn't. However, by using appropriate union definitions, generic procedures can be written.

C9. Variable Numbers of Parameters
Partially Satisfied

One can have a pointer to a linked list and pass the pointer as parameter or can have arrays of variable size, but, in general, a fixed number of arguments. Specific procedures, e.g. print, write, are present.

D. VARIABLES, LITERALS AND CONSTANTS

D1. Constant Value Identifiers
Satisfied

Use of = int K=15 is a constant

D2. Numeric Literals
Partially Satisfied

No specification for precision of real literals between object machines.

D3. Initialization
Satisfied

D4. Ranges and Stepsize
Not Satisfied

No subranges are built into the language.

D5. Variable Types
Satisfied

D6. Pointer Variables
Satisfied

Called ref variables, refer to objects of given type.

E. DEFINITION FACILITIES

E1. Type, Operator Definitions
Satisfied

New modes with operators; no encapsulation of modes with operators.

E2. Consistent Use
Satisfied

They may be used everywhere, but there are no built-in mechanisms for equal, read, print, etc. Existing operators may be extended for them.

E3. No Default Declarations
Satisfied

There is a standard prelude which explicitly makes a set of declarations and environment inquiries.

E4. Extend Existing Operators
Satisfied

E5. Type Definitions
Not Satisfied

No abstract types

E6. Data Definitions
Partially Satisfied

No power sets

E7. No Free Union
Satisfied

E8. Type Initialization and Allocation
Not Satisfied

F. SCOPES AND LIBRARIES

F1. Allocation and Access
Satisfied

F2. Limit Access Scope
Not Satisfied

No access limitations on mode declarations

F3. Static Scope Determination
Satisfied

F4. Libraries Available
Undetermined

F5. Library Contents
Undetermined

Implementation dependent - not ruled out by Reference.

F6. Libraries and Compools
Undetermined

F7. Machine Dependent Interface
Satisfied

The meaning of "special hardware" is unclear.

G. CONTROL STRUCTURES

G1. Control Structures
Partially Satisfied

Limited exception-handling, mostly confined to transput; no process control asynchronous interrupts.

G2. GOTO
Not Satisfied

Inadequate restrictions on target of the GOTO's.

G3. Conditional Control Structures
Partially Satisfied

The else part of an if...then is optional since fi brackets the if and eliminates the dangling else problem.

G4. Iteration Control
Partially Satisfied

Termination at loop beginning or via branch out of loop

G5. Recursive Routines
Partially Satisfied

Can define procedures within a recursive procedure and invoke them.

G6. Parallel Processing
Satisfied

Semaphores have operators down and up to permit mutual exclusion and synchronization.

G7. Exception Handling
Partially Satisfied

Limited number of transput exceptions

G8. Synchronization Primitives
Not Satisfied

Not specified in language; OS dependent

## H. SYNTAX AND COMMENT CONVENTIONS

H1. Syntax Characteristics
Partially Satisfied

Keywords (Indicants) can be abbreviated in some cases, e.g., ▼ for down.

H2. No Syntax Extensions
Partially Satisfied

Can define new operator priorities, new operators, but no new keywords.

H3. Source Character Sets
Satisfied

H4. Literals and Identifiers
Satisfied

H5. Continued Lexicals Units
Partially Satisfied

No character for end-of-line in literal strings

H6. Keywords
Partially Satisfied

Implementation dependent, requirement holds in reference language - some implementations reserve about 90 words.

H7. Comment Convention
Partially Satisfied

Comments may begin with comment, co, #, or ¢. They must terminate with the same symbol.

H8. Unmatched Parentheses
Satisfied

H9. Uniform Referent
Not Satisfied

[ ] for subscripts; ( ) for functions

H10. Consistency of Meaning
Satisfied

I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

I1. Defaults in Program Logic
Satisfied

I2. Default Override
Partially Satisfied

No programmer default override

I3. Compile-time Parameters
Partially Satisfied

Environment inquiries are part of standard prelude.

I4. Object Environment Compile-time Parameters
Not Satisfied

No conditional compilation - can discuss word size, etc.

I5. Language Kernel
Partially Satisfied

Not a particularly small base - TINMAN requires a relatively large base.

I6. Translator Restrictions Defined
Not Satisfied

No translator limits defined

I7. Object Environment Restrictions
Satisfied

## J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCE

J1. Efficient Code
Undetermined

Unknown - heaps, recursion, other features may be inefficient.

J2. Optimization Invisible
Not Satisfied

Evaluation order may change the value of an expression.

J3. Access to Machine Language
Not Satisfied

No machine language code insertions - ALGOL 68 not a systems implementation language - effect can be achieved by manipulation of bits and bytes. Note: there are no compile time conditionals.

J4. Implementation Representation Tradeoffs
Not Satisfied

J5. Open/Closed Implementations
Not Satisfied

No macro-definition; all closed routines.

# CHAPTER 6
# LIS

The contents of this chapter of the report involve material relating to LIS.

Section 1. Evaluation of LIS Against Language Design Criteria

Section 2. Evaluation of LIS Against the TINMAN

Section 3. Language Evaluation Summary for LIS

Section 4. Detailed Comparison of LIS to the TINMAN

# SECTION 1. EVALUATION OF LIS AGAINST LANGUAGE DESIGN CRITERIA

## 1. GENERAL COMMENTS

LIS, or "Language for Implementation of Systems" , clearly is a language oriented to implementation of "systems" programs. It "...belongs to the same stream of thought as the languages PASCAL, SUE, and SIMULA" (page 1). If LIS can be characterized at all, it is by this primary attribute: its intended use as a system implementation language.

There are many clues in the LIS definition that (quite naturally, given LIS's direction) corroborate this. A few of them are worth noting:

- LIS has no built-in REAL numbers or variables, either floating point or fixed point. Thus, if LIS is to be used for "conventional" programming, the language would have to have such types added.
- The built in data structures of LIS are oriented to (i) sets of objects, (ii) arrays of objects (i.e. ordered sets), and (iii) "plexes" which are structured record-type objects.
- The control structure in LIS is rich--perhaps "richer" than would otherwise be appropriate to a language as simple as LIS. This, too, seems to bias the suitability fo LIS towards complex "logical operations" as opposed to numeric computation.
- The inter-program communication structures provided by LIS are also quite rich. Besides the conventional routines format there is provision for recursion (in LIS PROCEDURES), and for co-routines. Surprisingly, however, there are no synchronization primitives built-in for LIS. These are easily handled with existing features, though.

Thus, LIS seems an able language for handling a problem that involves (i) much logical operation, (ii) relatively complex data structures, and (iii) integer/character oriented data. For an operating system as it conventionally perceived, this seems to be just the right mix.

LIS is regarded as a marginally satisfactory "base language" which might be amenable to extension to meet the specific requirements of the TINMAN.

## 2. ABSTRACTION

The degree to which LIS supports the notion of abstraction can be measured by, perhaps equivalently, investigating facilities within LIS that provide the ability to "hide" information. Overall, LIS scores reasonably well in this area, for the following reasons:

1. LIS requires a programmer to give complete descriptions of data and symbols relevant to each program unit. Consequently, "details" that a programmer would like to localize to one or more related program elements can be isolated there.

2. LIS provides a relatively rich inter-unit control structure, making it possible to pass "contexts" to a serving sub-program without having to know all of the details of what is done to the information in that sub-program.

3. LIS provides a limited capability for abstraction of data.

4. The uniform reference feature hides the representation of objects which may be treated in different ways.

Thus, LIS makes it possible (although, in some cases only with some programmer difficulty) to hide details from a programmer at a higher level.

On the other hand, LIS's capabilities do not extend to the definition of abstract objects (data structures and operations between them) by which it would be possible to fully encapsulate a part of a program solution. Part of the reason for this is that LIS's facilities for extension do not permit new operators and/or extension of existing operators to newly defined data abstractions. Hence, in this situation a programmer would have to resort to the more conventional method of

providing operators by writing sub-programs which support them on a one-by-one basis.

3. MODULARITY

LIS provides good support for modularity. This is achieved through traditional scoping rules associated through block structuring and by parameter passing rules which require parameter specifications to include the declaration of IN, OUT, or both to designate their use. Except with the use of direct code insertions, program modules cannot gain access to the body of other modules. This concept extends to data objects as well, through the uniform reference facility.

The detriments to modularity include the ability to insert machine code and the ability to use global variables within inner blocks, unless those variables have been declared PRIVATE.

In general, though, LIS includes the facilities needed to decompose problems into modules and to keep modules isolated throughout program construction.

4. LINEAR FLOW OF CONTROL

The control structures of LIS are sufficiently rich that it is easy to write programs which exhibit linear flow of control within a module and which conform to the one-in, one-out rules of structured programming. Since LIS has no GOTO statement, the language imposes a discipline of linear control flow. The programmer must be careful in using the rich control structures so that no unnecessary control structures are added. There is a definite possibility that extensive nesting of IF clauses could lead to excessive use of the EXIT statement from inner clauses, thereby defeating the purpose of the control structures.

## 5. CONTROL OF SCOPE AND BINDING

LIS variables are bound to a type and have a scope of allocation determined at compile time. Variables may be declared PRIVATE to a block so that they will not be regarded as global (i.e., accessible) in contained blocks. The variables may be further bound to a specific object representation through the USE facility.

All LIS variables are declared in this way, so that LIS has all of the desirable properties of scoping associated with block-structured languages. In this way, the LIS scope rules assist in supporting the language design goals of abstraction and modularity.

## 6. READABILITY AND COMPREHENSIBILITY

LIS programs are generally readable by the knowledgeable LIS programmer. However, the cues to understanding are somewhat different from those which one might use to read a PASCAL program. With PASCAL, for example, one has the tendency to look at choices of data representation. Given the LIS approach to data independence, one must look at the program logic first and suppress knowledge about data representations. This process initially goes against the intuition of jointly studying program and data structures.

LIS provides good commenting facilities and ample freedom for use of mnemonic variable and type names. The keywords are relatively few and are generally meaningful.

As with other system implementation languages, though, the programmer has the ability to create highly convoluted programs by extensive use of direct code, by use of the SET data type to do bit manipulation, and by large amounts of nesting of control structures and PLEX definitions. Since it is possible to write an elegant program in even the worst

of languages and a poor program in the "best" languages, there is no way to guarantee readable programs. However, LIS seems to encourage good programming style more than most other languages.

7. PROCEDURES AND DATA

LIS satisfies the requirement for a procedure to be deterministic. As LIS is defined there is no way for a program in LIS to modify itself, except by machine language insertions over which a language compiler has no control.

8. LANGUAGE SIZE

The LIS language is "small" in terms of the number of different syntactic constructions it provides (including built-in data types and operators). For instance, the "syntax flow diagrams" for LIS number only about 30; each one, of course, involves a number of different syntactic elements. There are only some 150 productions in the BNF definition of LIS, making it comparable to PASCAL in size.

9. ANALYZABILITY

Given a syntactically legal LIS program, how difficult would it be to analyze (as in an automated program analysis tool other than a compiler)? The answer to this question for LIS appears to be "not difficult at all." Several features of LIS support this:

1. The cleanness of the input format (blank delimited, reserved words, clarity in lexical units, etc).

2. The simplicity of program written in LIS (although not necessarily their length).

3. The limited range of applications to which LIS, as it presently is defined, seems appropriate (namely, "systems programs").

4. The elimination of the troublesome GOTO and labelled statements.

Thus, an automated program analyzer would have an easy time with LIS codes.

## SECTION 2. EVALUATION OF LIS AGAINST THE TINMAN

## 1. OVERVIEW

LIS supports a number of the features which are important in the creation of high quality systems software. With respect to the TINMAN, its strengths lie in the requirement for full declarations, the support that it provides for low level representations of objects, the control structures of the language, and the ability to unite separately compiled program units.

Because it was designed as a system implementation language rather than as a general purpose programming language, some of the features desired by TINMAN are missing. In particular, there is no mechanism for fixed or floating point numbers, a weak mechanism for multi-dimensional arrays, and very limited input/output facilities (no input at all).

Thus, it would be very difficult to write, for example, a matrix multiplication routine in LIS. Similarly, it would be difficult to write a variety of data processing applications programs involving use of sequential or direct access files (or any form of input at all). The strength of LIS lies in its utility for the construction of compilers and operating systems and for the degree of discipline which is enforced concerning type checking, use of parameters, etc.

LIS is a marginal candidate as a base language for TINMAN or for a language design predicated upon the requirements of the TINMAN. Its shortcomings are too great and the language addresses too narrow a range of applications. However, LIS contributes several features to programming language design which it appears to solve more effectively than any other language of which the project team is aware.

In particular, the PLEX is especially elegant. This feature alone provides a record-handling mechanism, permits

alternative structures to be declared in the record structure, and yields a uniform referent notation, by permitting objects within a record to be static structures or actions, i.e., to have their value computed by a function at run-time.

Another nice feature of LIS is the encapsulation of machine dependencies. While the inclusion of direct code runs counter to most of the other requirements of TINMAN, LIS protects against improper use of direct code more effectively than almost any other language, with the possible exception of revised (January, 1977) Euclid. LIS provides two levels of data independence rather than the one level which is common in other languages.

A third good quality in LIS is the collection of control structures, which includes spawning of independent processes.

In summary, then, the designers of a new programming language which includes system implementation as an application should study LIS carefully and should consider using the LIS approach for record structures (PLEXes) and for direct code insertions, regardless of the base language used for the language design.

## 2. DATA TYPES

LIS is a strongly typed language, requiring all procedures and data objects to be completely declared prior to compilation, and forbidding these types from being altered at run time (with the exception of machine code insertions).* While LIS supports integers, arrays, enumeration types, power sets of enumeration types, Booleans, and characters, it does not provide for

*There are numerous instances where the intent of the language design can be subverted by the insertion of direct code, e.g., to work with data as both integers and strings. It is assumed that the reader is aware of some of the potential problems posed by direct code. Hence, statements made about language characteristics will hereafter apply to the high-level language, with the understanding that any restrictions may be circumvented by use of machine code.

non-integer arithmetic. As noted above, LIS provides the concept of a PLEX to accomplish the requirements of a record-handling capability. The PLEX must be viewed as one of the most innovative contributions made by LIS. CASE statements embedded in the PLEX declaration permit records to have safely described alternative structures.

## 3. OPERATIONS

Assignment and reference operations are handled in the expected manner, with type checking in assignments. The relational operators are defined between any two simple expressions, where those simple expressions may designate, for example, an array or a single data element. The desired logical operators are present, except that XOR replaces nor. Scalar operations on arrays are nicely handled in LIS.

LIS does not permit side effects in functions, i.e., no parameter to a function may be altered, so that an implementor could implement short circuiting of logical operations without affecting the correctness of programs. (However, this feature is not specified in the language definition.) The abolition of side effects from functions, also followed in Euclid, GYPSY, and PLAIN, is regarded as essential to the fulfillment of the TINMAN objectives.

LIS fails to fulfill several of the objectives in this area, though. The absence of non-integer arithmetic means that several kinds of desirable operations are not present. While numeric subranges are handled, they are treated as separate types in the language declarations; the language reference manual is not explicit about checking of arithmetic expressions and operations involving objects of subrange type. Since subranges apply only to integer and enumeration types, the subrange objects use the same operations as those types.

LIS does not support input/output very effectively. There are no input facilities and very limited output facilities. Finally, the built-in operations on power sets are quite limited, although the existence of SET as a basic types, combined with function definition capability, makes it easy to provide operations (buy not binary operators) for union, intersection, difference, and complement.

4. EXPRESSIONS AND PARAMETERS

LIS is quite strong with respect to this set of TINMAN requirements. The number of levels of operator hierarchy is only four, the language is consistent with respect to the use of expressions, and the rules concerning parameter passing to functions eliminate side effects, so that the order of expression evaluation is not affected by optimization. LIS contains a good facility for the declaration and use of constants, although some constants must be evaluated at run time upon block entry. Type checking is performed between the formal and actual parameters on procedure calls.

LIS fails to meet some of the TINMAN requirements related to parameters, though. Parameters to subprograms may be of type value, result, or value-result. This technique, however, may not be used to pass the name of a subprogram as a parameter. LIS, instead, has a connector type for that purpose. There is no exception-handling mechanism in LIS and hence no parameter class for exceptions. LIS does not support generic functions (C8), nor are variable length parameter lists permissible.

5. VARIABLES, LITERALS, AND CONSTANTS

LIS provides the ability to associate constant values with different data types and to initialize variables of any type, although such initialization is not required. Since fixed point variables are not present in LIS, there is no provision

for specifying the range and step size of such variables.

LIS contains the REF mechanism as a pointer, which can be used to build data with shared and/or recursive substructures. The ROW facility provides a simplified way of indirectly pointing at an array.

## 6. DEFINITION FACILITIES

LIS does not provide for a data definition facility commensurate with the concept of abstract types. A programmer may define sets, plexes, and arrays, which are built upon simpler types, and may define the value of an object in a plex to be computed as an action, but there is no encapsulation of the object and its operators. It should be noted, though, that the ACTION mechanism comes close to what is desired. If one has declared

```
TYPE PERSON =
     PLEX
        BIRTH:  CONSTANT INTEGER;
          .
          .
        GETAGE: ACTION (AGE: OUT INTEGER);
          .
          .
     END;
```

then one may declare

```
JOHN: PERSON
```

and the use of JOHN.GETAGE provides some of the desired information hiding. However, there is no extension of existing operators to new types, and no facility for introducing new binary operators.

The definition facility, however, provides several ways for associating data objects with their defined type at compile time, including enumeration, array, and plex declarations. These have initialization procedures, but not finalization procedures.

## 7. SCOPES AND LIBRARIES

LIS permits variables to be declared PRIVATE so as to distinguish the scope of access from the scope of allocation. Private variables may not be accessed by blocks subordinate to the block in which the private variable is declared. Similarly, restrictions are placed on the visibility of data segments. The scoping rules of LIS are explicit and easily determined from the program structure at compile-time, as with other block-structured languages.

LIS provides a powerful facility for separate compilations, based on the presumed existence of libraries to store procedures and data segments. Furthermore, the implementation part of LIS envisions the inclusion of program segments written in languages other than LIS. These implementation parts may be included in LIS libraries along with procedure and data segments written in LIS itself.

LIS, however, does not include the facilities for machine independent interfaces to machine dependent capabilities, except in the most general sense, i.e., that machine language routines can be included.

## 8. CONTROL STRUCTURES

The LIS language has a powerful set of control structures, including LOOP-REPEAT, WHILE-LOOP-REPEAT, UNTIL-LOOP-REPEAT, a FOR statement, a FOR EACH statement, a CASE statement, an IF-THEN-ELSE, and subroutine calls and returns, including recursion. Subprograms may be invoked thorugh normal calls or through the use of connectors or references to objects in a plex where the object designates an action. Co-routines are also included in LIS.

With respect to the TINMAN requirements, the LIS syntax could be judged to have some shortcomings. However, there could easily be some debate as to whether failure to meet these

requirements are flaws in LIS of flaws in the TINMAN. For example, LIS permits nesting or recursive routines. Also, there is no exception handling mechanism. Finally, there is no real-time interrupt facility. Of course, TINMAN also requires a GOTO statement, which is not present in LIS.

On the whole, however, LIS control structures are among the best of all programming languages and give the programmer an excellent choice of control structures without any needless restrictions. The absence of the GOTO statement does not detract from programmability in any way. Specific characteristics of LIS which should be observed by the designers of a language to fulfill the TINMAN (or its successors) include the FOR statement, in which the control variable is treated as a locally declared variable, and the PROCESS mechanism, which permits the spawning of asynchronous processes. Parallelism is achieved by declaring a PLEX SHARED, in which WAIT and SIGNAL are attributes declared as ACTIONs, along with an integer semaphore. Thus, an instance of this plex may be associated with every shareable object in a system, thereby providing the facility for mutual exclusion. This capability is one of the more elegant features of LIS.

## 9. SYNTAX AND COMMENT CONVENTIONS

LIS programs are highly readable once the barrier of some of the new LIS constructs is overcome. LIS presents some features which are innovative in programming languages, and it takes a little experience before one becomes comfortable with these features. However, the complexity is considerably less than that of ALGOL 68. The readability of LIS programs is enhanced by the "prettyprint" processing of the translator.

The uniform reference notation is achieved in LIS (a major achievement), but has the disadvantage of detracting from program comprehensibility beyond a certain point. At a "high-level" reading of the program, uniform referents are

highly desirable, but when it is necessary to look at the lower levels of program construction, the loss of redundancy (compared with PASCAL) provided by the varying reference notations can be disconcerting. It is not immediately clear that the advantages of uniform referents (data independence) outweigh the disadvantages until one has considerable experience with their use. (This difficulty can be likened in some ways to the transition between file systems and data base management systems, in which many people experienced with file system were uncomfortable without knowing the representation of their data on secondary storage.)

The syntax is quite clean and free of duplications, except for the annoyance of some similarly constructed statements for assignments, initializations, and parameter results. = is used for both TYPE declarations and as a relational operator, initializations use ==, and input/output parameters use :=:. The parameter symbols must correspond with the IN and OUT notations of the formal parameter declarations; the use of two different notations, one in the procedure call and one in the procedure declarations, seem unnatural.

One cannot write self modifying programs in LIS. Everything is identifiable at compilation time. The syntax of the language cannot be changed and the operator hierarchies are immutable.

Even the break characters are present in LIS. The underscore character (_) may be used in the formation of names, e.g. IBM_370, and strings may be split, e.g., "HELLO" "THERE" is equivalent to "HELLOTHERE". There is no break character for integers, though.

Again, LIS rates very high on issues of language syntax and comment conventions.

## 10. DEFAULTS, CONDITIONAL COMPILATION, AND RESTRICTIONS

LIS does not provide defaults in program logic. Although there are defaults in object representation, the programmer has the opportunity to override those defaults through use of the USE statement.

The LIS language base is close to minimal, although there are several instances where there is more than one way to accomplish the same objective. For example, the FOR statement is inherently redundant, since it can be handled with a WHILE-LOOP. Also, the ROW feature could be adequately handled with the REF capability. There are a couple of other similar instances along this line, but they are relatively minor. LIS is well designed and hence has few unnecessary features; those duplications are given simply to make certain common operations easier.

## 11. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

The LIS programmer has access to direct code and to machine dependencies such as register allocation. This feature provides the ability to specify the object representation of composite data structures where desired. However, there is no "macro" facility in LIS which would permit both open and closed subroutines. Procedures, functions, and actions are all closed.

The language is moderate in size so that it should be possible to develop a reasonably efficient compiler. However, there are a few aspects of the language related to separate compilations and linkage of high-level data objects with their representations where some inefficiency may result. In particular, a programmer may specify a tightly packed object representation which would require the system to perform time-consuming packing and unpacking operations with each reference and each assignment to a data object.

## 12. PROGRAM ENVIRONMENT

LIS is intended to run in an operating environment which is capable of supporting asynchronous processes, subprogram libraries, and several other operating system facilities. To some extent, then, the requirement in TINMAN for parallel processing and asynchronous interrupt handling requires such an operating system. LIS features which necessitate a sophisticated operating system include the USE facility, the separate compilation ability, and the PROCESS declaration.

The language declaration does not make mention of the other tools available to the LIS programmer, but it is assumed that suitable text preparation, debugging, and testing facilities would be made available, since LIS would be an appropriate language for writing such programs.

## 13. TRANSLATORS

No standards have been established for LIS translators. To the knowledge of the project team, there is only one implementation of LIS. This implementation and the associated reference documents have been changing over the past several years, in keeping with typical language design efforts in which changes are made as experience is gained in use and implementation of the language. In that way, the existing compiler for the CII machine implements the language with no subset or superset implementation. It would be possible to write the LIS compiler in LIS.

Other aspects of translator characteristics are unknown. Because the project team is not informed as to the precise status of LIS translators, all of the evaluation criteria in this category have been left as undetermined. (This scoring may have detracted from the final relative score of LIS.)

## 14. LANGUAGE DEFINITION, STANDARDS AND CONTROL

LIS is innovative in several respects, so that it cannot be said that LIS is composed of proven features from other languages. However, the construction of a compiler for LIS demonstrates that the features of the language are well within the state of the art. Modifications are being made to the language and engineered into the translator.

As of yet, there is no formal definition for LIS along the lines of the W-grammar of ALGOL 68 or the proof rules which have been planned for Euclid. There were several instances within the language reference manual where the description was so terse or incomplete that the project team had to attempt to infer the intent of the language, with a lesser degree of certainty than was achieved for ALGOL 60 and ALGOL 68.

The language has been frozen recently, so it is possible that additional documentation will be forthcoming. We assume that some of this documentation is already available in French, but we have not had access to it.

Implementation and control of the language has been maintained until now by Jean Ichbian and CII. At this time, there appears to be only a single version of the source language so that another support agent could be appointed to manage the construction and validation of translators conforming to that language.

# SECTION 3. LANGUAGE EVALUATION SUMMARY FOR LIS

## HOL EVALUATION PROJECT:
## LANGUAGE EVALUATION SUMMARY FOR LIS

### A. DATA AND TYPES

A01 S
A02 N
A03 N
A04 N
A05 P
A06 N
A07 S

### B. OPERATIONS

B01 S
B02 N
B03 S
B04 S
B05 N
B06 P
B07 S
B08 N
B09 N
B10 N
B11 S

### C. EXPRESSIONS AND PARAMETERS

C01 S
C02 S
C03 S
C04 P
C05 P
C06 S
C07 P
C08 N
C09 N

### D. VARIABLES, LITERALS, AND CONSTANTS

D01 S
D02 S
D03 S
D04 N
D05 S
D06 S

### E. DEFINITION FACILITIES

E01 P
E02 S
E03 S
E04 N
E05 N
E06 P
E07 N
E08 N

### F. SCOPES AND LIBRARIES

F01 P
F02 P
F03 S
F04 U
F05 U
F06 P
F07 S

### G. CONTROL STRUCTURES

G01 P
G02 N
G03 P
G04 P
G05 P
G06 P
G07 N
G08 N

### H. SYNTAX AND COMMENT CONVENTIONS

H01 S
H02 S
H03 N
H04 P
H05 P
H06 S
H07 S
H08 S
H09 S
H10 P

### I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

I01 S
I02 S
I03 N
I04 U
I05 S
I06 U
I07 U

### J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCE

J01 U
J02 U
J03 S
J04 P
J05 N

### K. PROGRAM ENVIRONMENT

K01 S
K02 S
K03 U
K04 U
K05 S

### L. TRANSLATORS

L01 U
L02 U
L03 U
L04 U
L05 U
L06 U
L07 U
L08 U
L09 U

### M. LANGUAGE DEFINITION, STANDARDS AND CONTROL

M01 P
M02 S
M03 P
M04 U
M05 U
M06 U

Key: S = Satisfied
P = Partially Satisfied
N = Not Satisfied
U = Undetermined

# SECTION 4. DETAILED COMPARISON OF LIS TO THE TINMAN

## 1. INTRODUCTION

This section outlines specific evaluations of each TINMAN requirement as they are (or are not) met by the language LIS.

Page number references are stated for The System Implementation Language LIS, C.I.I., Document No. 4549 EL/EN, January 1976.

A. DATA AND TYPES

A1: Typed Language
Satisfied

LIS is fully typed (page 9 ff). Each entity must be declared and each such declaration must include a type specification. This is true both for built-in types and user-defined types (page 66 ff).

A2: Built-in Data Types
Not Satisfied

LIS provides predefined types for BOOLEAN, INTEGER, and CHARACTER (page 13 ff). In BOOLEAN, TRUE compares greater than FALSE. Fixed-point and REAL data types are not pre-defined. Arrays, sets, and record-structure data types are available. LIS's plexes are "records" with generalized and variant structure.

A3: Precision Declaration
Not Satisfied

Because neither REAL nor Fixed-point data declarations are predefined, correspondingly there is no precision specification possible. Appropriate type definitions are possible, however.

A4: Fixed Point
Not Satisfied

As per comments for A3, there exists no facility to indicate Fixed-point range and scale. Appropriate type definitions are possible, however.

A5: Enumeration Sense of Character Sets
Partially Satisfied

The predefined type CHARACTER in LIS is a symbolic type for which the character values are ordered according to the EBCDIC codes (page 13). USASCII is not mentioned.

A6: Arrays
Not Satisfied

The LIS language only has the ARRAY data type, but no predefined mechanism for n-dimensioned arrays, n < 1. Note, however, that 2- and higher-dimensioned arrays can be implemented by using an ARRAY...OF...ARRAY structure, although the accessing mechanism that results is somewhat clumsy.

A7: Record Equivalence
Satisfied

LIS's capabilities to define PLEXes makes it possible to imply record equivalence. Both whole records and/or parts of records can be equivalenced in this manner by using the CASE construction within a PLEX declaration. (Page 15 gives an example.)

B. OPERATORS

B1: Assignment
Satisfied

LIS provides for appropriate conversions between all built-in types (page 31).

B2: Equivalence Operator
Not Satisfied

Equivalence between any two entities of the same type is defined unambiguously in LIS (page 32); but this capability is not extended to differing types.

B3: Relationals Defined
Satisfied

The required (standard) relational operators are all pre-defined in LIS (page 30).

B4: Arithmetic Operations
Satisfied

Although the standard arithmetic operations are defined in LIS, because of the limited number of pre-defined types the language makes no provision for floating point operations.

B5: Truncation/Rounding
Not Satisfied

LIS, except by user extension, does not support control of truncation and rounding in arithmetic operations.

B6: Boolean Operations
Partially Satisfied

LIS includes the XOR operation rather than the NOR operation (page 30). The "short circuit" mode is not permitted, but could be implemented without difficulty.

B7: Scalar Operations
Satisfied

Scalar operations in LIS extend to composite data structures (page 38).

B8: Type Conversion
Not Satisfied

While LIS meets this requirement, the set of data structures predefined in the language makes the issue a very minor one. LIS has only INTEGER and CHARACTER types and so type conversions are not defined in a manner such as that required by the TINMAN.

B9: Range Exceptions
Not Satisfied

LIS has no provision for exception condition processing.

B10: I/O Operations
Not Satisfied

LIS has no input/output facility except that provided by symbolic output (page 50).

B11: Power Set Operations
Satisfied

C. EXPRESSIONS AND PARAMETERS

C1: Side Effects
Satisfied

LIS specifically indicates a left-to-right scan (page 31) for operations of the same precedence, but does not imply a left-to-right scan for all operations. Side effects, though, cannot occur.

C2: Operand Hierarchy
Satisfied

LIS predefined operators have a predefined hierarchy.

C3: Expressions Permitted
Satisfied

Expressions can be tolerated anywhere a simple variable can appear.

C4: Constant Expressions
Partially Satisfied

LIS does not indicate (one way or the other) how a constant expression is processed except to reassure the user that constants are constant. The CONSTANT attribute can be applied to any identifier. Not all CONSTANT values can be evaluated at compile time.

C5: Parameter Rules
Partially Satisfied

Parameter definition and use rules for LIS are consistent and unambiguous, but there is no provision for exception handling.

C6: Parameter Agreement
Satisfied

C7: Formal Parameter Kinds
Partially Satisfied

LIS provides for three kinds of formal parameters: read-only, read-write, write-only. The read-only parameter mode is the same as "call by value"; the read-write parameter mode is the same as "call by *reference-address*." The connector type is used to accomplish passing of procedure names as parameters.

C8: Formal Parameter Specifications Optional
Not Satisfied

LIS makes no provision for inferring parameter types by referencing an invocation.

C9: Variable Numbers of Parameters
Not Satisfied

Although LIS requires that the minimum number of actual parameters required be present, it makes no statement about what happens where an invocation has more than the minimum number of parameters (page 47).

D. VARIABLES, LITERALS AND CONSTANTS

D1: Constant Value Identifiers
Satisfied

In LIS's declaration structure there is explicit provision for initialization (page 9).

D2: Numeric Literals
Satisfied

D3: Initialization
Satisfied

D4: Ranges and Stepsize
Not Satisfied

LIS, by not explicitly providing for fixed-point data types, thereby makes no provision for range and stepsize specifications.

D5: Variable Types
Satisfied

E. DEFINITION FACILITIES

E1: User Type, Operator Definitions
Partially Satisfied

LIS provides for "implementation parts" (page 64 ff) which can define storage management properties for data types defined by the user. The orientation of this facility fo LIS is, however, specifically toward bit/byte/word/double-word manipulations of memory layouts. The language definition does not permit the addition of new operators (or operator symbols).

E2: Consistent Use
Satisfied

For types that can be defined, the usage rules in LIS are consistent. This results from the fact that only new data types, and not new operators, can be defined.

E3: No Default Declarations
Satisfied

E4: Extend Existing Operators
Not Satisfied

LIS provides no facility for extending the meaning of any of the existing operators, nor for defining new operators.

E5: Type Definitions
Not Satisfied

There is no abstract type definition in LIS.

E6: Data Definitions
Partially Satisfied

All of the operations mentioned in the TINMAN except discriminated union are possible.

E7: No Free Union
Not Satisfied

Free Unions can exist in LIS (page 12).

E8: Type Initialization and Allocation
Not Satisfied

LIS gives no facility for defining the processing of new types and thus has no facility for intialization and/or allocation.

## F. SCOPES AND LIBRARIES

F1: Allocation and Access
Partially Satisfied

LIS provides for inclusion of one entity in another and defines the scope rules for inner-defined quantities.

F2: Limit Access Scope
Partially Satisfied

LIS provides no general capability for limiting access. PRIVATE variables give a limited capability to control the scope of access to a variable.

F3: Static Scope Determination
Satisfied

LIS, being a well-structured and fully typed language with a requirement that all entities be fully declared, intrinsically accomplishes deterministic scoping.

F4: Libraries Available
Undetermined

LIS makes no mention of a program library facility; however, there are provisions for separate compilation which imply such a library.

F5: Library Contents
Undetermined

LIS includes no facility for accessing a compile-time library.

F6: Libraries and Compools
Partially Satisfied

LIS includes data segments which appear to resemble compools. They may be used in a general program structure.

F7: Machine Dependent Interface
Satisfied

LIS makes it possible to incorporate in-line sequences of machine language text (page 73). The layout used, and the associated rules strongly resemble System 360/370 coding conventions and register use rules.

G. CONTROL STRUCTURES

G1: Control Sturctures
Partially Satisfied

LIS provides for the standard structured porgramming control structures, with the addition of an "ELSE IF" element in the selector construct. LIS supports co-routine calls (page 47). LIS has no explicit facility for exception handling, or asynchronous interrupt handling. Note that co-routines could be used to implement the required synchronization primitives and parallel processing relatively easily.

G2: GOTO
Not Satisfied

LIS has no labelling facility and no GOTO statement.

G3: Conditional Control Structures
Partially Satisfied

LIS provides for Boolean condition selection and includes a CASE statement that provides a computed choice.

G4: Iteration Control
Partially Satisfied

LIS provides for termination anywhere in a loop (page 45), provides for subtyping of a value (either symbolic or literal) in a CASE-like statement (page 42), and, by default, assures top-entry to iteration since there is no capability for labelling statements. Control variables in LIS are handled properly.

G5: Recursive Routines
Partially Satisfied

LIS provides for recursive procedures (page 3), but it is also possible to define a procedure within a recursive procedure.

G6: Parallel Processing
Partially Satisfied

LIS provides the capability to allocate resources (FREE and ALLOCATE keywords, page 49) relating to PLEX definitions. LIS does not provide for explicit control of parallel processing.

G7: Exception Handling
Not Satisfied

LIS, in the language definition, makes no mention of exception handling, although it seems to be envisioned by Ichbiah.

G8: Synchronization Primitives
Not Satisfied

LIS has no built-in synchronization primitives. However, because LIS has co-routines and plexes with actions, these could be implemented rather easily.

H. SYNTAX AND COMMENT CONVENTIONS

H1: Syntax Characteristics
Satisfied

The "clarity and comprehensibility" of LIS, a judgemental issue, is helped on two features: the capability to reparagraph the source text, and the capability for commenting.

H2: No Syntax Extensions
Satisfied

LIS provides for a fixed syntax.

H3: Source Character Set
Not Satisfied

LIS programs are defined in terms of a subset of EBCDIC (page 6), while the CHARACTER Mode operates on the full EBCDIC character set (page 13).

H4: Literals and Identifiers
Partially Satisfied

There is a break facility for strings. The underscore can be used in names. There is no break character in numbers.

H5: Continued Lexicals Units
Partially Satisfied

LIS requires fully juxtaposed lexical tokens. There is no End-of-Line Symbol, however.

H6: Keywords
Satisfied

LIS has 82 reserved keywords.

H7: Comment Conventions
Satisfied

LIS's comment convention (page 8) has easy to use and understand processing rules.

H8: Unmatched Parentheses
Satisfied

H9: Uniform Referent
Satisfied

H10: Consistency of Meaning
Partially Satisfied

LIS provides no capability to redefine the meaning of a language-defined symbol, but "=" is used both in type definitions and as a relational operator.

## I. DEFAULTS, CONDITIONAL COMPILAITON AND LANGUAGE RESTRICTIONS

I1. Defaults in Program Logic
Satisfied

LIS, which requires full specification on the part of the programmer, supports the property wanted by the TINMAN.

I2. Default Override
Satisfied

LIS has no defaults.

I3: Compile-time Parameters
Not Satisfied

I4: Object Environment Compile-time Parameters
Undetermined

I5: Language Kernel
Satisfied

LIS in its entirety is small enough to be considered as its own kernel, although there is some redundancy.

I6: Translator Restrictions Defined
Undetermined

I7: Object Environment Restrictions
Undetermined

## J. EFFICIENT OBJECT REPRESENTATION AND MACHINE DEPENDENCE

J1. Efficient Code
Undetermined

J2. Optimization Invisible
Undetermined

J3. Access to Machine Language
Satisfied

This is permitted in LIS (page 73 ff).

J4. Implementation Representation Tradeoffs
Partially Satisfied

LIS provides a limited facility for specifying machine layouts (page 65 ff), but refers only to bit/byte/word/double-word layouts in memory. This facility appears oriented to providing protection against storage reference mistakes due to machine changes.

J5. Open/Closed Implementations
Not Satisfied

The programmer in LIS has no control over the form of implementation of a subroutine/subprogram.

K. PROGRAM ENVIRONMENT

K1. Operating System Not Required
Satisfied

K2. Program Assembly
Satisfied

LIS's description includes the possibility of separately compiled modules.

K3. Software Development Tools
Undetermined

K4. Translator Options
Undetermined

K5. Assertion Capability
Satisfied

LIS can be made to provide for assertions treated as comments.

## L. TRANSLATORS

All of the TINMAN issues in regard to the translator for LIS are "undetermined", primarily for lack of detailed information about the LIS translator.

## M. LANGUAGE DEFINITION, STANDARDS AND CONTROL

M1. Existing Features
Partially Satisfied

LIS contains a number of unique language features, including the PLEX implementation specifications, and attribute definitions, but all of these seem implementable in a straightforward way.

M2. Language Subsets
Satisfied

The LIS manual provides verbal explanations of the semantics of each LIS construct for which the interpretation could be different from the common assumptions. However, the description is not given in a rigorous semantic description format.

M3. Language Documentation
Partially Satisfied

To the extent that the LIS manual represents a tutorial, this requirement is satisfied.

M4. Control Agent
Undetermined

M5. Support Agent
Undetermined

M6. Library Standards
Undetermined

# CHAPTER 7
# CS-4

The contents of this chapter of the report involve material relating to CS-4.

Section 1. Evaluation of CS-4 Against Language Design Criteria

Section 2. Evaluation of CS-4 Against the TINMAN

Section 3. Language Evaluation Summary for CS-4

Section 4. Detailed Comparison of CS-4 to the TINMAN

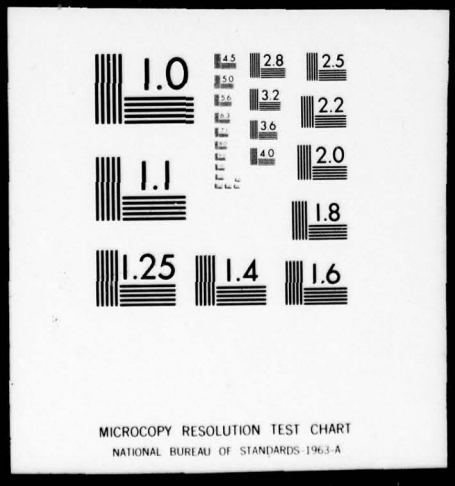
# SECTION 1. EVALUATION OF CS-4 AGAINST LANGUAGE DESIGN CRITERIA

AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

7 OF 18
ADA037634

7 OF 15
ADA037634



## 1. INTRODUCTION

This section examines the extent to which CS-4 satisfies the criteria for a "good" higher-order language as presented in "Criteria for Evaluation of High-Order Languages." CS-4 is seen to fulfill the needs moderately well. However, there are certain weaknesses in most of the areas, caused by the size of the language, the number of inconsistencies in the syntax and semantics, and the number of untested language features. CS-4 is marginally suitable for consideration as a base language for the TINMAN, but is a much weaker candidate than is ALGOL 68 or PASCAL.

## 2. ABSTRACTION

CS-4 provides features for procedural abstraction through functions and procedures. A function is a procedure with a result specification. Procedures accept a variety of parameters: these parameters fall into major classes: call by value, call by reference, and call by name. Call by value is identical to the PASCAL notion, call by reference also follows PASCAL and ALGOL 68, call by name follows ALGOL 60. Each of these three classes is further distinguished by designating whether a parameter is input, output, or both. An additional parameterization facility is provided by the ATTRIBUTE concept, which permits exceptions to be passed.

There are some problems associated with permitting both call by name and call by reference. Call by name is intended to be used mainly for passing the name of a procedure as a parameter, but is not restricted in any sense. Thus, one may pass algebraic expressions (Jensen's device) and (it would appear) a constant by name just as well. Ledgard has shown (Computing Surveys, Sept., 1971, pp. 126-7) some of the potential for confusion when both call by reference and call by name are permitted.

CS-4 also has a facility for data abstraction. However, the parameter passing rules for data abstractions are different from those of procedures. Some of the better features of the CS-4 data abstraction facility are the ability to designate the names which are accessible outside the mode definition (CAPABILITY) and the protection given to local names. Furthermore, the set of standard mode operations is useful and provides a degree of equivalence among data types. The ability to specify finalization (TERM) actions is unique to CS-4. On the other hand, though, one can object to the facility for mode generators incorporated within CS-4. One of the legal parameter types for a mode declaration is a mode itself, i.e., a mode is treated as a mode. This presents some implementation difficulties, defers the binding of modes of objects, detracts from the clarity of programs, and seems to run counter to the language philosophy of strict type checking.

Other suitable abstraction facilities are provided through a sizeable number of built-in data types, including integers, reals, booleans, arrays, records, strings, sets, union types, complex numbers, vectors, matrices, fractions, and enumeration types (STATUS). All of these built-in types have a set of associated operations. This large number of built-in types is a major contributor to the size of the CS-4 language, but makes the language suitable for a broad set of applications.

## 3. MODULARITY

CS-4 is a block structured language in which variables may be passed between blocks. A procedure with local declarations is considered a block for this purpose. A good deal of modularity is presented by the block structure and the parameter passing mechanism. Furthermore, CS-4 supports separate compilation of program modules and contains ACCESS and RENAME facilities to use desired names or eliminate conflicts between

names in separately compiled modules.

There are, though, some ways to subvert the intended modularity. One is through the use of global variables, which may be used within contained blocks. Access to names contained within a block may, as usual, be kept invisible to external blocks. Also, there is no control over side-effects in functions, as there is in such languages as Euclid and Gypsy. CS-4 leaves the order of expression evaluation undefined, so that it is possible to subvert the intent of modularity by having a module (that which implements the function) affect the computational result of another module by side effect.

In general, though, one can achieve modularity in CS-4 as well as with other modern programming languages, e.g., PASCAL. It is simply necessary to be careful in module construction and in the use of global variables and procedures.

## 4. LINEAR FLOW OF CONTROL

CS-4 has a number of different control structures which are intended to eliminate the need for unrestrained transfers of control. In addition to the procedure invocation, there is an IF statement, a CASE statement (including OTHERWISE), and the usual FOR, REPEAT, and WHILE constructions. The FOR statement can assure that the control variable is local to the scope of the loop. The GO TO statement prohibits transfers to inner blocks, but permits transfer out of several nested levels with a single command, contrary to the strictures of good programming style. A variant of the GO TO is the ABORT TO.

CS-4 also has an exception-handling mechanism which can be used to provide alternate flow of control in the event of some condition arising, possibly a user-defined condition. The SIGNAL statement (or the automatic occurrence of an exception) causes a procedure to be invoked and permits parameters to be passed. The SIGNAL may be passed to successive levels of

handlers with a RESIGNAL statement, a construct which is unknown and untested in other languages. It would appear that there is some danger of setting up a completely alternative program flow through exceptions and parameters which is completely different from the visual program structure. Once again, there are adequate facilities for writing a good program, but too many additional facilities which may lead to the construction of poor programs.

5. CONTROL OF SCOPE AND BINDING

CS-4 is similar to ALGOL 60 in this respect, in that the language has a block structure. The differences are that CS-4 has STATIC allocation (somewhat similar to ALGOL 60 own) and SHARED allocation for multiple processes, which can be protected or unprotected. SHARED allocation is somewhat like FORTRAN COMMON in that objects may be shared among modules with relatively little control; this can be used to destroy the intent of modularity, but is seen as being relatively useful for efficiency. (CS-4 programmers should be encouraged to use structured data and pass such objects as parameters by reference; a single parameter produces relatively little overhead.)

With the exception of SHARED objects, the scope of variables is as in ALGOL and PASCAL, with the block structure determining the scope of access. The only way to prevent inner blocks from accessing names declared in other containing blocks is by renaming those variables in the inner blocks.

CS-4 sets up a mechanism intended to provide a rational approach to scope and binding, then turns around and gives the programmer the ability to override that mechanism with the SHARED feature. In addition, the CS-4 programmer has the ability to make machine code insertions in programs, giving the ability to circumvent virtually all of the intended restrictions.

## 6. READABILITY AND COMPREHENSIBILITY

CS-4 is a large language by any measure. It is virtually devoid of defaults, so that even the most obvious attributes must be declared. For example, one cannot simply declare an integer (with range determined by implementation and undefined initial value), but must explicitly include a range and initialization. These requirements lead to a rather wordy program, which offers certain advantages and disadvantages with respect to readability. Some of the benefits in CS-4 are the ability to associate KEYWORDS, i.e., formal parameters, with actual parameters in a procedure call, a good commenting facility, no arbitrary restrictions on identifier length, and powerful abstraction facilities. It is clear that one can write a readable, if somewhat verbose, program in CS-4.

With respect to comprehensibility, though, CS-4 seems to rate somewhat lower. The complexity of the language is such that few people will completely understand all of its mechanisms. There are so many inconsistencies, such as the parameter passing rules dependent on what is being declared (procedure or mode). There are a vast number of built-in exception names. There are nearly 300 reserved words, in three different flavors, some of which can be overridden. There is a long list of built-in operators for each built-in type. Some of the syntactic constructs do not seem particularly logical, especially those in procedure headings. It is the rare programmer who will become fully conversant with the language to the extent that most programmers are comfortable with languages such as ALGOL 60, FORTRAN, or even COBOL.

The composition of correct CS-4 programs seems to require a great deal of attention to detail and memorization of a large number of syntactic forms. The number of untried or rarely used features in CS-4 makes it difficult for a novice to learn and use the language and makes it difficult for those experienced

with other, smoother languages to become comfortable with CS-4. In many respects, it has the complexity of PL/I, but none of the defaults. It would appear that CS-4 would be among the most difficult of languages to learn (surpassing even ALGOL 68) and among the easiest to forget. The training investment for CS-4 would be considerable.

## 7. PROCEDURES AND DATA

The high order features of CS-4 makes a firm distinction between procedures and data. However, the ability to include machine code insertions eliminates the protection provided by this high level feature and permits the construction of self-modifying programs in the direct code sections. There is apparently no mechanism available which can prevent such an occurrence as long as one is permitted to use machine code; furthermore, there is no guarantee that these machine code insertions will limit their effects to a single module. The machine code could obtain the name of another procedure, load that name, hence obtaining its address, and make modifications within that procedure!

## 8. LANGUAGE SIZE

By most common measures, CS-4 is a very large language. It contains over 400 productions in the basic language syntax, excluding such things as I/O which are not part of the base language. There are more than 250 nonterminals in the syntax definition. There are nearly 300 keywords. The language is not merely large or baroque; it is best described as rococo! ("excessively ornate or intricate")

This size has serious implications not only for learning (as noted above) but for implementation. The language is presently being implemented for the first time on a large-scale computer. The feasibility of creating an efficient implementation of CS-4 on a small machine is highly questionable. There are some untested language features whose implementability

are not well known.

## 9. ANALYZABILITY

CS-4 programs lend themselves to analysis quite well except under error conditions, in which the control structure may become quite complex. There are only a couple of trouble spots, caused by the multi-level exits provided by the GO TO and ABORT TO statements. The RANGE feature of the language will automatically trap those variables which leave a certain predefined range. An interface to the operating system will permit various timing operations to be performed quite easily as well.

## 10. SUMMARY

The criteria used here are fundamental to language design and underlie much of the current research. While other criteria could easily be added, one can obtain a good feeling for the potential of the language by using this list alone.

One's initial impression of CS-4 is that it satisfies TINMAN quite well and that it tries to address many of the issues raised in TINMAN. The impression becomes increasingly worse as one studies the language, particularly in comparison with other research languages currently under development. The conclusion can only be that CS-4 makes some useful contributions to language design and achieves some language features which are missing from other languages, but the resulting whole leaves much to be desired.

## SECTION 2. EVALUATION OF CS-4 AGAINST THE TINMAN

## 1. OVERVIEW

CS-4 is being designed to conform to the requirements for programming languages being established by the DoD, most recently expressed in the TINMAN. Although it comes relatively close to the TINMAN on a strict evaluation of conformance to requirements, there are a number of problems with the language.

Its greatest strength from the standpoint of TINMAN is the support given to various built-in and extended data types, its exception handling mechanism, its ability to incorporate object representations for specific machine implementations, and its range/initialization facilities. Its greatest weaknesses are lack of recursion, lack of a pointer mechanism, absence of input/output and parallel processing from the base language, and lack of alternative structures in records.

There are many inconsistencies in the language, such as in the kind of types which may be used as parameters in different places. One is not given the impression of a language which was carefully designed as a synthesis of good language features. Instead, one receives the impression of starting with a basic set of language structures and adding features in ad hoc way as new requirements were added to the TINMAN, or as a clearer understanding of the TINMAN requirements was gained. The language lacks the smoothness and orthogonality of ALGOL 68 or PASCAL, which are built upon a more solid foundation involving a degree of formalism. CS-4 incorporates too many little-understood concepts.

Although there are some features of CS-4 that are handled particularly well and which could be incorporated into the language which is designed (or adapted) to meet the TINMAN (as revised), CS-4 does not seem to be a suitable base language for that effort. The language is already unwieldy, with far too many statements, keywords, and syntactic structures.

Adding the capability for recursion, pointers, and other "essential" language features would almost certainly lead to a language which could neither be used nor understood.

Part of the difficulty lies with the TINMAN rather than with CS-4 itself. A separate section "Notes on the TINMAN Requirements," points out some of the inconsistencies and problems with the TINMAN. It is to be expected that any language which attempted to meet all of the requirements would of necessity fail.

(It should also be observed that most well designed programming languages are the result of the work of one, or at most two, individuals. This is the case with ALGOL 68, PASCAL, LISP, APL, SNOBOL 4, etc., all of which, whatever their strengths or weaknesses, have made a contribution to programming languages. Those languages which have been designed by committee, with the notable exception of ALGOL 60, have the tendency to grow large and inconsistent without an adequate underpinning as the result of trying to meet the diverse views of their designers.)

## 2. DATA AND TYPES

CS-4 is a fully typed language. Built-in types include INTEGER, REAL, STATUS, FRACTION, STRING (FIXED and VARYING), SET, COMPLEX, and BOOLEAN. These may be formed into ARRAYs and STRUCTUREs. Union types may be formed. A one-dimensional array of reals may be declared as a VECTOR; a two-dimensional array of reals may be declared as a MATRIX. Both VECTOR and MATRIX types have a built-in set of operations associated with them.

There are no fixed point variables, violating TINMAN requirements A4 fully and A2 in part. Alternative structures in CS-4 are handled by declaring an object of UNION type within a structure along with the appropriate TAG field.

Declarations of objects are quite thorough, including a range part and an initialization part. Whereas one might declare

int a

in ALGOL 68, a comparable declaration in CS-4 is

VARIABLE A IS INTEGER (RANGE: MIN-PORTABLE-INTEGER-VALUE THRU MAX-PORTABLE-INTEGER-VALUE) [NOVALUE];

Array declarations are handled similarly. Arrays may have any number of dimensions and may be of any mode, including ARRAY. A6 is not entirely met, though, since an array may be passed to a procedure with the upper and lower bounds designated by *, with the value supplied at block or procedure entry time.

3. OPERATIONS

CS-4 defines the meaning of assignment for every type, including a mechanism whereby assignment can be defined for user-defined types. However, the treatment of union types is not entirely consistent with this. In order to make an assignment within a union type, it is necessary to make certain that the tag of the union type corresponds to the types of the value which is to be assigned.

CS-4 considers the range specifications of an object as part of the type definition. In other words, two objects are of different types if their ranges are different. In that case, a run-time check must be made on the value to be assigned to make certain that it is within the range of the variable receiving the value. Otherwise an exception is signaled.

The "=" operator only applies for objects of the same type. One good feature of CS-4 with respect to relational operators is the capability to compare real numbers within a

precision. Since the declaration of reals includes a precision (possibly defaulted), the concept of equality, with respect to floating point numbers, takes on a different meaning. This distinction is quite significant in applications involving numerical computation where successive rounding errors may lead to an unexpected result, such as an extra iteration through a loop.

Scalar operations are not defined for real arrays (B7) and there is no way to extend those operators to achieve this objective. However, scalar operations are defined for objects of type VECTOR and MATRIX, i.e., for special one and two dimensional objects of type real. One may, in general, though make data transfers between structures or arrays with identical logical structures.

Another inconsistency in CS-4 shows up in the integer/character conversion. The ASCII function converts characters to integer but there is no inverse function for converting numerics to character.

No input/output operations (B10) are in the base language; all are in the operating system interface for the language.

4. EXPRESSIONS AND PARAMETERS

CS-4 does not specify the order in which expressions are evaluated other than to say that operators with the highest precedence are evaluated first. Since there is no restriction of side effects within functions, The ordering of expression evaluation could produce different results than would left-to-right. Since the language does not specify this ordering, various implementors can act accordingly, making whatever optimizations they see appropriate. This seems to be a rather serious omission from the language definition, especially since portability of programs is a goal both of CS-4 and TINMAN. (C1).

Another language inconsistency arises in conjunction with passing parameter. The parameters to a procedure are those which satisfy production 28 for primitive-mode-invocation, (p. 211) while the parameters for a mode declaration are those which satisfy production 206, restricted-mode-invocation (p. 223). It is not clear why the restriction was imposed for mode declarations.

Parameter passing itself is somewhat inconsistent as well. There are 10 classes of parameter passing, which may be grouped into 4 categories. There are three forms of parameter passing by COPY (similar to call by VALUE), three forms of parameter passing by REF (copy by REFERENCE), and three forms of parameter passing by NAME (copy by NAME). "Copy by Name" permits procedure and function names to be passed; it also permits expressions to be passes as in ALGOL 60 and thus leads to a mechanism for Jensen's device. Most of the binding attributes are straightforward except for COPYO, in which the actual parameter is <u>not</u> used to initialize the copy in the formal parameter; instead, such a parameter must be initialized. The 10th class of parameter passing is the attribute HANDLES, which accepts 1 or more exception names. C7 is thus not completely satisfied.

CS-4 requires the full description of all formal parameters. There is no default situation as suggested by C8 which would permit generic procedures; in fact, generic procedures are hard to construct in CS-4. It is not immediately clear as to the restrictions on union types being passed as parameters. A formal parameter may be of union type, but may not have a binding attribute of REFI, REFO, or REFIO. Apparently, though, it can be passed by NAME! It is not clear as to whether an object of UNION type may be passed as an actual parameter to a formal parameter having one of the constituent types or whether an object of simple type, being one of the constituents of a union type, can be passed as an actual parameter to a formal parameter of a UNION type of which the simple type is one of the constituent types.

This must be cleared up in the language semantics.

CS-4 has no mechanism for variable arguments, although the input/output procedures accept them. One cannot in general define procedures with such a facility. A restricted means for doing this would be with union types. No other techniques are immediately apparent since there is no pointer mechanism; one could create an artificial pointer mechanism using integer variables (or the equivalent) and then pass the integer indicating the beginning address of a linked list, but this is a rather clumsy solution.

5. VARIABLES, LITERALS, AND CONSTANTS

CS-4 meets these requirements quite well, save for the absence of fixed point variables (D4) and pointer mechanisms (D6). CS-4 provides a full range of initialization and range capabilities, including an initialization procedure INIT which will be defined by the user for extended types. The range and initialization parts are required for all variable declarations. The major objection is that they can be quite cumbersome to use in simple programs.

6. DEFINITION FACILITIES

CS-4 programmers have access to strong definitional facilities, including those of abstract data types. The user may define structures and unions of existing types, may incorporate those structures or unions in more complex types, including other structures and unions, and may associate a set of procedures and/or functions with the representation of an object so that the representation of the object is hidden.

Mode declarations are actually type generators, since a type can be passed as a parameter to a mode declaration. One could declare the mode stack and then declare objects

which are integer stacks, real stacks, or even stacks of stacks. (There does not appear to be a restriction on recursive mode definitions, even though there is a restriction on recursive procedure definitions!) These mode declarations include built in operation names ASSIGN, INIT, COMPARE, and TERM, which provide facilities for assignment, initialization, comparison, and finalization of objects of the defined type. The user may define these as desired. The set of names which is visible from outside the mode declaration is given in the CAPABILITY statement. (CAPABILITY is a rather unusual choice of word, since it has a different explicit and well-known meaning with respect to operating systems; EXPORTS seems like a more appropriate word to denote the names of operators which are exported from the mode declaration and made known outside the declaration.)

Unfortunately, the existing dyadic operators cannot be extended to these new types, nor is there a method for defining new infix operators. All use of user-defined types is through function and procedure calls, thereby violating E2.

CS-4 appears to be headed in the right direction here, but Euclid appears to have solved this set of problems more effectively, although it, too, fails on the required operator extension feature.

## 7. SCOPES AND LIBRARIES

CS-4 facilities with respect to libraries and compools is undetermined. These features are dependent upon the environment in which the program executes. The mechanism for external procedures seems well-suited for libraries, though.

Variables may be declared STATIC or AUTOMATIC or SHARED (with or without protection), with AUTOMATIC as the default case. Scope is determined at compile time for

STATIC and AUTOMATIC variables. The scope of a SHARED variable is determined within a procedure, but its scope includes other programs as well.

The mode declaration mechanism provides a set of capabilities (i.e., names) which can be used to access an object of a user-defined mode. F1 and F2 are met.

F7 is not met since there is no input/output in the base language.

8. CONTROL STRUCTURES

CS-4 has all of the basic control structures, but has no facilities for recursion, parallel processing, or asynchronous interrupt handling. However, parallel processing and interrupt handling can be accomplished through the operating system interface of the language.

CS-4 meets the requirements G2 and G3 partially. There are inadequate restrictions on the go to, since one can branch into an enclosing block as well as within the most local scope. Furthermore the EXIT statement, which is used to terminate loops, may branch through more than one nested loop construct, thereby terminating more than one loop. G3 is almost met as well, but the IF-THEN-ELSE is not fully partitioned, since there is no requirement that the ELSE be included. No ambiguity results, since the IF statement is terminated with a FI, whether or not the ELSE appears. The statement IF A THEN B FI may be seen as being equivalent to the fully partitioned statement IF A THEN B ELSE ; FI where the ";" represents a null statement.

The CS-4 exception-handling mechanism is reasonably powerful and well-controlled. Procedures are associated with exceptions and are named as handlers. The appropriate handler for a given exception is determined by dynamic scoping rules. Approximately 60 signals are built into CS-4 to which the

user may append an arbitrary number of user-defined signals. G7 is well met and the flow of control upon exceptions is well-restricted. CS-4 meets requirement G7 as well as any existing language, and significantly better than PL/1 does.

## 9. SYNTAX AND COMMENT CONVENTIONS

CS-4 is not a simple language. It thus fails to fully satisfy requirements H1 and H6. The grammar of the base language contains over 400 productions, with approximately 250 non-terminal symbols! A significant number of these productions exist as a result of inconsistencies in the language among the syntax forms. While the language can be parsed in a straightforward way, the tree is quite large and the language could not easily be implemented on small-to-medium sized computers. The language defines more than 250 keywords, which fall into three categories. The first category is 48 words with fixed meanings. Others are either words which have meaning to CS-4 in limited places only or which can be freely overridden by users. This is in contrast to most other languages, with the notable exception of PL/I.

H8, the restriction on parentheses, is met, except that a programmer might use a character string such as "ABC(", which contains an unmatched parenthesis. The syntax, though, does not permit unmatched parentheses.

As with most other languages, CS-4 fails to achieve a uniform reference mechanism. The structure addressing syntax uses a "." notation, while function calls, mode definitions, and array subscripts use parentheses. However, array elements could be replaced by function calls.

While there are few unique notations for special cases, the syntax of the language occasionally becomes quite complex, particularly when one is defining arrays or structures as parameters within procedures.

## 10. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

The CS-4 language meets this set of requirements quite well. There are facilities for providing machine structure definitions when the programmer wishes to override the default definition of a STRUCTURE type. Characteristics of the host machine may be associated with the program at compile time so that storage alignment and size can be properly worked out. This feature applies to STRUCTURE variables and not to other program types. However, one could declare a single variable within an MSTRUCTURE to achieve the desired effect.

The language base houses most of the power of the language. However, input/output and parallelism are in the operating system interface rather than in the language itself. The language base is far from being minimal, since there are often many alternative ways of doing the same thing; VECTOR and ARRAY of REAL with one dimension is one example. The size of the language and the rather poor presentation of the language in the reference document detract significantly form the understandability of the language. The language semantics are relatively poorly defined.

## 11. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

It is difficult to determine the potential efficiency of the language translator. The size of the grammar would appear to impose a significant compilation cost. Checking ranges and checking parameters of union type might impose run time costs not generally incurred with other languages. For example, one could write

```
VARIABLE I IS INTEGER (RANGE: 1 THRU 10) [1],
         J IS INTEGER (RANGE: 1 THRU 10) [1];
```

where the compiler might not see that there is no need for a run time check on an assignment of I to J or vice versa. One only hopes that the compiler would not build in run time checks on:

```
VARIABLE (I,J) IS INTEGER (RANGE: 1 THRU 10) [1];
```

It is quite likely though that the first instance would be treated as two different types simply because of the disjoint declarations.

The effect of optimizations (J2) is unknown in the absence of information about the ordering of expression evaluation and the prevention of side effects (C1).

CS-4 meets the other TINMAN requirements in this category. MPROCEDURES, MSTRUCTURES, and the OPEN and MOPEN features give the desired abilities. OPEN and MOPEN provide a generalized MACRO facility.

12. PROGRAM ENVIRONMENT

CS-4 requires an operating system interface since there is no input/output in the base language (K1). An operating system interface is necessary for integration of separate modules into the program; a requirement staisfied by the EPROCEDURE concept. EPROCEDURES permit not only CS-4 programs, but programs written in other languages, to be integrated with CS-4 programs. CS-4 provides a built-in capability for PL/I and FORTRAN as other HOL's, and a capability to specify routine linkage mechanisms for other calling protocols.

CS-4 is intended to run in an environment which provides useful tools for program creation and debugging. It should be noted, though, that the tools on the IBM 370, which is the initial target machine, leave a great deal to be desired.

Assertions may only be included as comments. CS-4 is not intended for verification. Indeed, there are no proof rules for CS-4 itself and no formal specification of the language semantics. There undoubtedly will be many surprises for CS-4 programmers as they determine how the language processor handles some of the language issues which are not clearly specified in the reference document.

## 13. TRANSLATORS

Very little can be said about CS-4 translators. The language is sufficiently large that one is tempted to eliminate certain features in a first implementation. Furthermore, minimization of compile time as an objective is difficult to achieve, for reasons which have been taken up in the section "Notes on the TINMAN Requirements."

It is expected that a CS-4 translator can be written for a number of different host computers, but there is some question as to whether such a translator can run on a relatively small computer. Some form of cross-compiler, as anticipated by L5, might be needed.

It is impossible to address L6 clearly, although it is apparent that the language provides the necessary features for full type and range checking.

It would be feasible to use CS-4 as an implementation language for a CS-4 translator (L9).

## 14. LANGUAGE DEFINITION, STANDARDS, AND CONTROL

TINMAN criterion M1 is not well met. Many of the features of CS-4 are not well understood; abstract data types, for example, have not been fully worked out. Many of the language notions, including FRACTION, RESIGNAL, MOPEN, and NORECALL, are only lightly used or are untried elsewhere. The impression is that these features were often added as patches to a previously designed language.

Criterion M2 is not well met. There is no unambiguous definition of semantics; much is left to interpretation, requiring the reader to be familiar with a large number of programming language concepts. Some semantic issues are not addressed; e.g., expression evaluation and parameter passing involving union types are incompletely explained.

Criterion M3 is partially met. There is a CS-4 primer, but there is not a "formal in-depth description." The CS-4 Language Reference Manual does not meet that requirement.

How well CS-4 meets the requirements of DoD control over translators, standards, and libraries is impossible to state. The developer of the language or the contracting agency could impose a sufficient set of controls to achieve the objective, particularly since the present implementation level of the language is still very low.

# SECTION 3. LANGUAGE EVALUATION SUMMARY FOR CS-4

HOL EVALUATION PROJECT:
LANGUAGE EVALUATION SUMMARY FOR CS-4

A. DATA AND TYPES

A01 S
A02 P
A03 S
A04 N
A05 P
A06 P
A07 S

B. OPERATIONS

B01 P
B02 N
B03 S
B04 P
B05 S
B06 S
B07 P
B08 P
B09 P
B10 N
B11 S

C. EXPRESSIONS AND PARAMETERS

C01 N
C02 S
C03 S
C04 P
C05 N
C06 S
C07 N
C08 N
C09 N

D. VARIABLES, LITERALS, AND CONSTANTS

D01 S
D02 S
D03 S
D04 P
D05 S
D06 N

E. DEFINITION FACILITIES

E01 S
E02 N
E03 S
E04 N
E05 P
E06 S
E07 S
E08 S

F. SCOPES AND LIBRARIES

F01 S
F02 S
F03 S
F04 U
F05 U
F06 U
F07 N

G. CONTROL STRUCTURES

G01 P
G02 P
G03 P
G04 S
G05 N
G06 N
G07 S
G08 N

H. SYNTAX AND COMMENT CONVENTIONS

H01 P
H02 P
H03 S
H04 S
H05 S
H06 N
H07 N
H08 S
H09 N
H10 S

I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

I01 S
I02 P
I03 S
I04 N
I05 P
I06 S
I07 S

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCE

J01 U
J02 U
J03 S
J04 S
J05 S

K. PROGRAM ENVIRONMENT

K01 N
K02 S
K03 U
K04 U
K05 P

L. TRANSLATORS

L01 U
L02 U
L03 U
L04 U
L05 U
L06 S
L07 S
L08 S
L09 S

M. LANGUAGE DEFINITION, STANDARDS, AND CONTROL

M01 N
M02 N
M03 S
M04 U
M05 U
M06 U

Key: S = Satisfied
P = Partially Satisfied
N = Not Satisfied
U = Undetermined

# SECTION 4. DETAILED COMPARISON OF CS-4 TO TINMAN

## 1. INTRODUCTION

This section outlines specific evaluations of each TINMAN requirement as they are (or are not) met by the language CS-4.

Page number references where given refer to the basic reference document for this language: CS-4: Language Reference Manual and Operating System Interface, Intermetrics, Inc., IR-130-2, October 1975.

A. DATA AND TYPES

A1. Typed Language
Satisfied

A2. Built-in Data Types
Partially Satisfied

No fixed point

A3. Precision Declaration
Satisfied

A4. Fixed Point
Not Satisfied

No fixed point

A5. Enumeration Sense of Character Sets
Partially Satisfied

STATUS types handle this. EBCDIC not specified.

A6. Arrays
Partially Satisfied

Both lower and upper array bounds may be undetermined at compile time.

A7. Record Equivalence
Satisfied

Alternative structures may be created by declaring UNION types in structures.

B. OPERATIONS

B1. Assignment
Partially Satisfied

Assignment to an object of union type can only be from another object of union type. There are also restrictions concerning assignments when the ranges of two objects do not intersect.

B2. Equivalence Operator
Not Satisfied

The objects compared must agree in type.

B3. Relationals Defined
Satisfied

There are also functions for relations on reals to perform comparisons within a precision.

B4. Arithmetic Operations
Partially Satisfied

No fixed point

B5. Truncation/Rounding
Satisfied

B6. Boolean Operations
Satisfied

B7. Scalar Operations
Partially Satisfied

No scalar operations on conformable arrays. Scalar operations on REAL VECTORS. Data transfers between records or arrays of identical logical structure are permitted.

B8. Type Conversion
Partially Satisfied

No fixed point. ASCII function converts characters to integers, but there is no inverse. An actual parameter must be the same type as a formal parameter. If the formal parameter is of union type, the actual parameter (p. 146) must be of the same union type, not a constituent type.

B9. Range Exceptions
Partially Satisfied

No fixed point. X-INTEGER-RANGE may be signaled.

B10. I/O Operations
Not Satisfied

The base language has no input/output operations.

B11. Power Set Operations
Satisfied

C. EXPRESSIONS AND PARAMETERS

C1. Side Effects
Not Satisfied

C2. Operand Hierarchy
Satisfied

3 levels of relational, 4 levels of arithmetic hierarchy

C3. Expressions Permitted
Satisfied

C4. Constant Expressions
Partially Satisfied

Evaluation of constant expressions cannot be determined.

C5. Parameter Rules
Not Satisfied

Parameters exist for exception-handling, procedures, and declarations. The types of parameters that can be passed are different for each of the classes. Also, different kinds of binding are associated, with only the NAMEI binding class being meaningful for mode-invocations (pp211, 222).

C6. Parameter Agreement
Satisfied

C7. Formal Parameter Kinds
Not Satisfied

There are 9 binding attributes: COPYI, COPYO, COPYIO, NAMEI, NAMEO, NAMEIO, REFI, REFO, and REFIO. The HANDLES attribute passes exception names. This could be satisfied if, for example, COPYI, COPYO, and COPYIO were considered to form a class. However, objects other than procedure names may be passed by NAME.

C8. Formal Parameter Specifications Optional
Not Satisfied

C9. Variable Numbers of Parameters
Not Satisfied

D. VARIABLES, LITERALS, AND CONSTANTS

D1. Constant Value Identifiers
Satisfied

D2. Numeric Literals
Satisfied

D3. Initialization
Satisfied

D4. Ranges and Stepsize
Partially Satisfied

No fixed point variables

D5. Variable Types
Satisfied

D6. Pointer Variables
Not Satisfied

No pointer mechanism

E. DEFINITION FACILITIES

E1. Type, Operator Definitions
Satisfied

E2. Consistent Use
Not Satisfied

There is no facility for extending operators to new types.

E3. No Default Declarations
Satisfied

E4. Extend Existing Operators
Not Satisfied

E5. Type Definitions
Partially Satisfied

There is some inheritance of data operations.

E6. Data Definitions
Satisfied

E7. No Free Union
Satisfied

E8. Type Initialization and Allocation
Satisfied

F. SCOPES AND LIBRARIES

F1. Allocation and Access
Satisfied

Can have STATIC allocation

F2. Limit Access Scope
Satisfied

The operations associated with a separately defined structure are specified in a capability list. Only those names are available outside the mode declaration.

F3. Static Scope Determination
Satisfied

F4. Libraries Available
Undetermined

F5. Library Content
Undetermined

F6. Libraries and Compools
Undetermined

F7. Machine Dependent Interfaces
Not Satisfied

G. CONTROL STRUCTURES

G1. Control Structures
Partially Satisfied

No recursion, parallel processing, or interrupt handling in base language

G2. GOTO
Partially Satisfied

There is a goto, but the branching scope includes enclosing blocks as well as the most local scope.

G3. Conditional Control Structures
Partially Satisfied

The else is not required following an if then.

G4. Iteration Control
Satisfied

G5. Recursive Routines
Not Satisfied

No recursion (p. 140)

G6. Parallel Processing
Not Satisfied

G7. Exception Handling
Satisfied

(See p. 154)

G8. Synchronization Primitives
Not Satisfied

## H. SYNTAX AND COMMENT CONVENTIONS

### H1. Syntax Characteristics
Partially Satisfied

The grammar is context free, but quite large, with over 400 productions involving approximately 250 non-terminals.

### H2. No Syntax Extensions
Partially Satisfied

Some of the key words may be redefined, as identifiers or procedure names for example.

### H3. Source Character Set
Satisfied

### H4. Literals and Identifiers
Satisfied

For identifiers (__) and numeric literals (^) (p. 9)

### H5. Continued Lexicals
Satisfied

(p. 5)

### H6. Keywords
Not Satisfied

CS-4 has approximately 260 Keywords, divided into three categories. Category 1 are fixed. Category 2 have fixed meaning in certain contexts, but may be freely used elsewhere. Category 3 may be overridden by programs. There are 48 Category 1 Keywords (pp 261-3).

### H7. Comment Convention
Not Satisfied

There are two comment conventions: (| |), which may be nested and may cover multiple lines, and % which treats the remainder of a line as comment.

H8. Unmatched Parentheses
Satisfied

The only exception is perhaps within string literals.

H9. Uniform Referent
Not Satisfied

The structures use a . notation, which is different from ( ) used by procedures and arrays.

H10. Consistency of Meaning
Satisfied

## I. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

I1. Defaults in Program Logic
Satisfied

I2. Default Override
Partially Satisfied

The programmer may declare MSTRUCTURES for defining object representations at the machine representation level.

I3. Compile-time Parameters
Satisfied

I4. Object Environment Compile-time Parameter
Not Satisfied

No conditional compilations in source language

I5. Language Kernel
Partially Satisfied

VECTOR largely duplicates REAL ARRAY, and MATRIX largely duplicates REAL ARRAY with 2 dimensions. The base language, given its size, is not "simple."

I6. Translator Restrictions Defined
Satisfied

I7. Object Environment Restrictions
Satisfied

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCE

J1. Efficient Code
Undetermined

Depends on parsing techniques

J2. Optimization Invisible
Undetermined

J3. Access to Machine Language
Satisfied

MPROCEDURES (pp.194-5)

J4. Implementation Representation Tradeoffs
Satisfied

MSTRUCTURES

J5. Open/Closed Implementations
Satisfied

OPEN attribute for procedures

K. PROGRAM ENVIRONMENT

K1. Operating System Not Required
Not Satisfied

Language assumes OS interface for I/O, etc.

K2. Program Assembly
Satisfied

EPROCEDURES

K3. Software Development Tools
Undetermined

K4. Translator Options
Undetermined

K5. Assertion Capability
Partially Satisfied

Included only as comments

L. TRANSLATORS

L1. Language Supersets
Undetermined

L2. Language Subsets
Undetermined

L3. Low-Cost Translation
Undetermined

L4. Many Object Machines
Undetermined

L5. Self-Hosting
Undetermined

L6. Translator Checking
Satisfied

L7. Diagnostic Messages
Satisfied

CTWARNINGS option

L8. Translator Structure
Satisfied

L9. Self-Implementable
Satisfied

It would be possible. However, the absence of recursion and pointers detracts from the usability of some of the favored parsing methods.

## M. LANGUAGE DEFINITION, STANDARDS AND CONTROL

M1. Existing Features
Not Satisfied

Many language notions are lightly used or untried elsewhere. CAPABILITY, FRACTION, RESIGNAL, MOPEN, NORECALL are just a few of these.

M2. Unambiguous Definition
Not Satisfied

The present language definition is not entirely clear on semantics. For example, order of expression evaluation is not specified.

M3. Language Documentation
Satisfied

M4. Control Agent
Undetermined

M5. Support Agent
Undetermined

M6. Library Standards
Undetermined

# CHAPTER 8
# REFERENCES

1. W.A. Whitaker, "Department of Defense Requirements for High Order Computer Programming Languages, Tinman", 1 March 1976.

2. P. Naur (Editor), "The Revised Report on the Algorithmic Language ALGOL-60". Communications ACM, January 1963.

3. J.B. Goodenough and L.H. Shafer, "A Study of High Level Language Features". SofTech/ECOM, ECOM-75-0373-F, Volumes 1 and 2.

4. J.D. Ichbiah, J.P. Rissen, and J.C. Heliard, "The Two-Level Approach to Data Independent Programming in the LIS System Implementation Language". Compagnie Inter. Pour L'Informatique, Louveciennes, France, June 1975.

5. The System Implementation Language LIS, CII. Document No. 4549 EL/EN, January 1976.

6. CS-4 Language Reference Manual and Operating System Interface. Advanced Software Technology Division, NELC, San Diego, California, October 1975.

7. CS-4 Primer, Volume I, Basic Features. Advanced Software Technology Division, NELC, San Diego, California, February 1975.

8. A.I. Wasserman, "Programming Language Designs and Programming Discipline". UCSF, Medical Information Science, San Francisco, November 1976.

9. A.I. Wasserman, "PLAIN: Reliable Interactive Software and Programming Language Design." Technical Report No. 20, UCSF, Laboratory of Medical Information Science, January 1977.

10. A. Van Wijngaarden, et. al., "Revised Report on the Algorithmic Language ALGOL 68". Acta Informatica, Volume 5, pages 1-236, 1975.

11. A.S. Tanenbaum, "A Tutorial on ALGOL 68". Computing Surveys, Volume 8, No. 2, June 1976.

12. R. Baumann, M. Feliciano, F.L. Bauer, and K. Samuelson, Introduction to ALGOL. Prentice-Hall, Englewood Cliffs, New Jersey, 1964.

13. J.D. Ichbiah, J.C. Heliard, "PLexes in LIS," 1975.

14. J.J. Ichbiah and P. Cousot, "Visibility and Separate Compilations," 1975.

15. Evaluation of ALGOL-68, JOVIAL J3B, PASCAL, SIMULA 67, and TACPOL vs. Tinman Requirements for a Common High Order Programming Language, Softech, Inc. Report No. 1021-14, October 1976.

16. R. Roessler and K. Schenk (Editors), "International Purdue Workshop on Industrial Computer Systems: A Language Comparison," Purdue Laboratory for Applied Industrial Control, Purdue University, October 1976.

17. R.M. De Morgan, I.D. Hill and B.A. Wichmann, "A Supplement to the ALGOL 60 Revised Report," The Computer Journal, Volume 19, No. 3, 1976.

18. C.M. Geschke and J.G. Mitchell, "On the Problem of Uniform References to Data Structures," 1975 International Conference on Reliable Software, pages 31 ff, April 1975.

19. Hoare, C.A.R., "Hints on Programming Language Design," Stanford University Computer Science Department CS-73-403, December, 1973.

# EVALUATION OF

# CORAL 66

Prepared by

RLG Associates, Inc.
11250 Roger Bacon Drive
Reston, Virginia 22090

November 18, 1976

(5)

This report presents an evaluation of the CORAL66 language with the requirements listed in the "TINMAN". (DOD Requirements For High Order Computer Programming Languages, "TINMAN" - 1 March, 1976, Section IV). For purposes of comparison CORAL66 is considered to be defined by:

Official Definition of CORAL66 Third Edition, 1974

There are 78 language requirements listed in Section 4 of the "TINMAN". This report compares CORAL66 with each individual requirement. A summary of the degree to which the language satisfies each requirement is presented.

The introductory paragraph of each "TINMAN" requirement is included as the leading section for each requirement evaluation.

Symbols placed beside each individual requirement indicate the degree to which the language meets the requirement. The symbols and their meanings are as follows:

T - Totally meets requirement

P - Partially meets requirement

F - Does not meet requirement

U - Unknown from available documentation

A summary of the evaluation is included as the last section of this report. Merits and failures of the language as it currently is defined are discussed. Also discussed are the potential language modifications that can be made to support DOD "TINMAN" requirements.

## A. DATA AND TYPES

Requirement: A1

The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source program.

---P---

In general, CORAL66 is a typed language. Most types are known at compile time. However, two specific deficiencies exist in this area. Pointers are not defined explicitly as pointers but rather as integers. Floating point variables are declared as integer and use a software language extension to perform the necessary arithmetic operations.

Additionally, the OVERLAY construct provides a method for violating type checking and verification.

Addition of the keyword POINTER to the language, with appropriate changes to the syntax, will provide a method of uniquely identifying pointer variables. For detailed information see (D6).

Similarly, extension of the language to require specific typing of floating point numbers can be accomplished.

The OVERLAY option can be restructured within the language but the impact on existing programs or future programs operating on limited machine configurations may preclude this option. Development of a method to permit an overlay operation and also preserve type checking appears to be very expensive.

---

Requirement: A2

The language will provide data types for integer, real (floating point and fixed point), boolean and character and will provide arrays (i.e., composite data structures with indexible components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

---P---

Integer.....T
Fixed point.....T
Floating point.....F
Character string.....T
Boolean.....F
Arrays.....P
Records.....P

Floating point -- Floating point variables are declared as integer type in the language. Floating point operations are implemented as procedure calls to library routines.

Boolean -- There are no boolean variables and no bracketed boolean expressions in the CORAL66 language.

Arrays -- Arrays are provided for within the language. However, arrays are limited to two dimensions.

Records -- There is a provision for record structures within the language. Records are definable by a table mechanism. However, records do not define types, nor may they be used as type constructors.

The language can be modified to include both floating point and boolean capabilities at modest cost. Expansion of array and record capabilities, however will prove to be more expensive.

---

Requirement: A3

The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

---F---

The language does not include this capability. Inclusion of an optional single floating point precision declaration within a block that pertains to all local floating point identifiers will require an additional language keyword, PRECISION, which would be used as follows:

```
BEGIN
FLOATING PRECISION 1E-4 ;
FLOATING A, B, C ;
```

The following rules would apply:

1) All arithmetic operations on local (to a scope) floating variables will be truncated to the specified precision.

2) Operations on floating data between local and non-local identifiers will be truncated to the lower precision.

The requirement to enable individual specification of identifiers would still not be met. Precision specification would be considered to pertain to all floating identifiers declared within a given block.

---

Requirement: A4

Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.

---T---

CORAL66 totally meets this requirement.

---

Requirement: A5

Character sets will be treated as any other enumeration type.

---F---

A printing character is assumed to have a unique integer representation dependent on some hardware or software convention. The form is included within the integer syntax definition. Printing characters are implementation dependent. Layout characters may not be included as arguments of literals.

---

Requirement: A6

The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

---P---

Number of dimensions fixed at compile time.....T
Type fixed at compile time.....T
Lower subscript bound fixed at compile time....T
Subscripts from contiguous range.....T
Subscripts from enumeration type.....P
Upper subscript bound fixed at scope entry.....F

Subscripts are confined to integer type. The language does not allow an explicit character as a subscript. See A5 for further information.

Arrays are limited to two dimensions and are fixed at compile time. Addition of variable upper bounds to arrays will require extensive change to the language definition.

---

Requirement: A7

The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified

by the user at compile time.

---p---

Alternative structures are permitted by the use of the overlay feature and by use of pointers. Components must be specified at compile time. However, type checking and array bound protection are not secure. There are no facilities included within the language to identify type disagreements or array overruns or underruns. Inclusion of a discrimination feature will require extensive restructuring of the formal definition of the language, and indeed may not be economically feasible.

---

## B. OPERATIONS

Requirement: B1

Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

---p---

Variable declaration for all data types.....T
Encapsulated type declaration.....F
Array or record declaration.....P

Assignment is by individual variable or by individual entry within tables (records). The language does not permit assignment of complete arrays or records. These aims may be approached by definition of user written procedures but still will not provide a consistent language definition and notation.

---

Requirement: B2

The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

---p---

The source language can be used to compare any two data objects for identity. The comparator works on a bit by bit basis without regard to type. This means that, conceivably, reals, integers and characters could be considered equivalent.

Type comparison, within the constraints of the "TINMAN",

for equivalence add a new dimension of difficulty. Limited type comparisons, when the absolute value of subscripts are known or when a specific equivalence relationship is explicitly stated, can be achieved. However, use of pointers and overlay features for element access provide an escape mechanism from type equivalence checking that cannot be completely avoided. Practically speaking, either the rules for type checking must be relaxed or the constructs permitted in the language must be severely restricted.

---

Requirement: B3

Relational operations will be automatically defined for numeric data and all types defined by enumeration.

---p---

All six relational operators are included. Unordered sets are not included in CORAL66.

---

Requirement: B4

The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

---p---

Addition.....T
Subtraction.....T
Multiplication.....T
Division (with real result).....T
Negation.....T
Exponentiation.....F
Division with remainder.....F

Exponentiation is not supported in the language. Remainder after division is not explicitly available in the language.

Division with remainder can be included by the addition of a special remainder operator and the appropriate syntax modification.

Exponentiation capabilities can also be included.

---

Requirement: B5

Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond

the specified precision will be allowed for floating point numbers.

---U---

Truncation rules are not explicitly discussed in the formal definition of the CORAL66 language.

The assumption is made that most of the truncation rules specified by this requirement are met since they are good compiler development practices.

---

Requirement: B6

The built-in boolean operators will include "AND", "OR", "NOT", and "NOR". The operations "AND" and "OR" will be evaluated in short circuit mode.

---P---

Short circuit "AND".....T
Short circuit "OR".....T
NOT.....F
NOR.....F

The operations "NOT" and "NOR" are not included within the language. Short circuit mode evaluation for scalars is defined within the language.

"NOT" and "NOR" operators can be added to the language at very modest cost.

---

Requirement: B7

The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

---F---

Assignment between records and arrays.....F
Scalar operations on arrays.....F

The language only supports element by element access of arrays and records. The capability can be added to the language. Again, type checking for some operations can be easily included but use of pointer and overlay capabilities can still provide a method to circumvent complete type checking.

---

Requirement: B8

There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit

conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

---P---

Explicit conversions.....T
No implicit conversions.....F
Explicit between scale factors.....T

The programmer can impose any desired system of evaluation by the use of Number type (Expression). If no explicit conversion rules are specified, implicit language rules are used for conversion. Changes to eliminate the implicit conversion capability will have minimal impact.

---

Requirement: B9

Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

---F---

The language does not support this function.

The capability for run time exception checking can be included within the language.

---

Requirement: B10

The base language will provide operations allowing programs to interact with files, channels or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

---F---

The language does not explicitly define I/O operations. All operations are accomplished by calls to procedures which are tailored to the required hardware configuration. Language constructs can be defined to satisfy this requirement. There are however, complications and machine dependency problems. Bit manipulation constructs provided by the language can be used to generate programs that are machine dependent. While this violates "TINMAN" constraints, this capability is necessary in many applications.

---

Requirement: B11

The language will provide operations on data types de-

fined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

---F---

Power sets are not defined in the language. This capability can be added with not too much difficulty.

---

## C. EXPRESSIONS AND PARAMETERS

Requirement: C1

Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

---P---

Although an algorithm for evaluating expressions does not form part of the formal definition of the language, all syntactically outermost terms in an expression will be evaluated to the required numeric type before the adding operators are applied. However, if an expression is enclosed in round brackets, its terms are no longer 'outermost' and the rule no longer applies. In this case the

algorithm for the particular compiler determines the sequence of events. Additionally, the programmer may impose any desired system of evaluation.

---

Requirement: C2

Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

---P---

Unambiguous presentation.....P
Few precedence levels.....T
Require explicit parentheses.....F
No user defined levels.....T

Precedence levels are those normally encountered in all languages. The language can be modified to require parentheses in certain cases if the list of cases can be agreed upon by all users.

---

Requirement: C3

Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

---T---

There are no special case constraints on expressions included in the syntax of the language.

---

Requirement: C4

Constant expressions will be allowed in programs where constants are allowed, and constant expressions will be evaluated before run time.

---T---

The requirement is completely met by the language.

---

Requirement: C5

There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types for exception handling, for parallel processes, for declaration, or for built-in operators. There will be no special operations (e.g., array structuring) applicable only to parameters.

---P---

Consistency in parameter rules.....P

No special parameter operations.....T

Exception handling and parallel processing are not explicitly addressed within the language.

The formal parameter and the passed parameter must agree in type according to language rules. However, as was the case in requirement A1, this type checking can be defeated by use of pointers and computed array or table location values.

---

Requirement: C6

Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

---P---

Dimensions determined at compile time.....T
Size and subscripts can be passed.....F
Type agreement for actual and formal parameters.....P

In general there is agreement of type between formal and actual parameters. Use of computed subscripts and use of pointer capabilities still provide methods for defeating type checking in all cases.

The language can be modified to pass array size and subscripts as arguments and the capability to support more than two dimensional arrays can be included.

---

Requirement: C7

There will be four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

---P---

Parameter constants.....T
Parameter variables.....T
Exception conditions.....P
Procedure parameters.....T

The constructs VALUE, LOCATION, and LABEL identify the input, output and exception handling capabilities of the language. Exception control is supported by use of label parameters. There is no formal classification of exception parameters specified within the language.

---

Requirement: C8

Specification of the type, range, precision, dimension, scale and format of parameters will be optional on the formal side. None of them will be alterable at run time.

---P---

Optional properties.....P
Fixed at run time.....F

Range and dimension of arrays and tables cannot be specified within procedure definition. All other properties must be specified.

It is not apparent to the reviewer how this requirement can be achieved and the requirements of A1 and A2 simultaneously achieved; perhaps a special construct, GENERIC, to be used only by an elite, tightly controlled group of programmers.

Support of this capability requires a significant change to the syntax of the CORAL66 language.

---

Requirement: C9

There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

---F---

Variable number of arguments.....F
All but constant number same type.....F
Number fixed at compile time.....T

The number of arguments for all procedures is fixed at compile time. In one sense, passing of a formally declared array name as an argument provides access to a variable number of arguments. This apparently is not the intent of this requirement but could be judiciously used to partially satisfy it.

The syntax of the language can be changed to support this requirement but the magnitude of the change indicates an almost complete redefinition of segments of the language.

---

## D. VARIABLES, LITERALS AND CONSTANTS

Requirement: D1

The user will have the ability to associate constant values of any type with identifiers.

---T---

The language provides this capability.

---

Requirement: D2

The language will provide a syntax and a consistent interpretation for literals of built-in data types. Numeric literals will have the same value (within the specified precision) in both programs and data (input or output).

---P---

Literals of built-in data types.....P
Consistency in value.....P

Floating point numbers cannot be compiled since the compiler works only in fixed point. The existing documentation is unclear as to whether the values of literals are a function of compile time, run time or both.

---

Requirement: D3

The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

---P---

Declare initial values.....P
Allocation scope initialization.....P
No default values.....T

Certain data objects may be initialized when the program is loaded into storage by use of the presetting clause in the data declaration. Presetting is not dynamic, and preset values which are altered by program execution are not reset unless the segment or program is reloaded. An object is not eligible for presetting if it is declared anywhere within

1) the body of a recursive procedure or,
2) an inner block of the program or,
3) an inner block of a procedure body.

Requirement: D4

The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

---F---

There is no capability specified within the language for range checking.

The compiler can be expanded to include range checking mechanisms. Checks for compile time usage can be included with relatively little cost. Run time checks, however, will require an extensive overhaul of compiler syntax.

---

Requirement: D5

The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.

---F---

The language does not provide this capability. It can be added as a subset of the potential additions discussed in D4. The language has a very limited array/table capability. Only two dimensional arrays are permitted. There is limited capability for the interrelation of arrays and records. A major restructuring of the language will be necessary to meet the intent of the "TINMAN" requirements.

---

Requirement: D6

The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

---P---

Shared/recursive substructure capability.....T
Variable/record/array component handling.....T
Typed pointer characteristic.....F
Allocation never wider than access.....F

The pointer capability provided by CORAL66 is wide open. Virtually no checking of type or access scope is possible. Pointers are defined as integers and thus may take on any value permitted an integer. It is possible to access any core location by mistake using the pointer.

In order to satisfy "TINMAN" requirements, several addi-

tions must be made to the syntax of the compiler. First, the pointer must be typed explicitly as such. Next, the rules governing pointer use must be defined. In this case pointers should be limited to record or array structures with a series of restrictions limiting the range of values the pointer may take. These restrictions should preclude computational assignment of value to the pointer. Values should be limited by the compiler to that set which is completely known at compile time.

Obviously this will severely limit the power and usefulness of pointers but stringent type checking and scope checking demand this limitation.

---

## E. DEFINITION FACILITIES

Requirement: E1

The user of the language will be able to define new data types and operations within his programs.

---F---

This requirement is not supported by the language. New data types may be introduced implicitly by definition of procedures to perform the function of a new data type. In this case, however, type checking and allocation scope checking rules can easily be violated.

---

Requirement: E2

The "use" of defined types will be indistinguishable from built-in types.

---F---

The language does not permit definition of new data types as an operation.

This capability can be added to the language as an extension of E1.

---

Requirement: E3

Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

---T---

CORAL66 completely meets this requirement.

---

Requirement: E4

The user will be able, within the source language, to extend existing operators to new data types.

---F---

The language does not permit new type definition. This capability can be included as part of the extension to type definition.

---

Requirement: E5

Type definitions in the source language will permit definition of both the class of data objects comprising the types and the set of operations applicable to that class.

A defined type will not automatically inherit the operations of the data with which it is represented.

---F---

The language does not support user defined types.

---

Requirement: E6

The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

---P---

Enumeration.....F
Cartesian products.....T
Discriminated union.....F
Power set of enumeration type....F

The table mechanism is used to define records. Its use is limited, however. Records do not define types and cannot be used as type constructors.

Arrays, as has been mentioned before, are limited to two dimensions.

The language may be modified to support this requirement. Table and array structures are very limited within the language and should be expanded.

---

Requirement: E7

Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.

---P---

Does not permit free unions.....F
Does not permit subsetting.....T

Free unions are permitted with use of pointer and overlay functions within the language.

Subsetting is not defined within the language.

---

Requirement: E8

When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

---F---

Types cannot be defined within the language.

---

## F. SCOPES AND LIBRARIES

Requirement: F1

The language will allow the user to distinguish between scope of allocation and scope of access.

---P---

Initialization of variables is accomplished by a common communicator used by the system. Initialization is performed only once under the rules stated in D3.

Out of scope access to variables is possible when using the pointer facility of the language.

---

Requirement: F2

The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

---F---

As previously stated, the language does not support definition of new data types.

The language does not distinguish between normal structures and "special" structures. The language can certainly be modified to include such a capability but it will be costly.

---

Requirement: F3

The scope of identifiers will be wholly determined at compile time.

---T---

This requirement is completely met by the language.

---

Requirement: F4

A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

---P---

Variety of data and operations.....P
Easily accessible.....T

The language documentation mentions a number of procedures and routines that are available. Whether the libraries contain all things desired by this requirement is not determined.

The cost of providing a "variety" can range from modest to exhorbitant, depending on what functions are considered necessary.

Available routines are easily accessible to the programmer.

---

Requirement: F5

Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

---P---

Accessible at compile time.....T
Other source language routines.....P

The language does not preclude routines written in a different source language. As long as the routines exist in an object form compatible with CORAL66 they can reside in system libraries. There is no provision for interface checking in the language.

---

Requirement: F6

Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

---P---

Libraries and compools indistinguishable.....F
Hold anything definable in language.....T
Many levels of access.....T
Many specialized subsets.....P

The segments of a program may communicate with each other through a common global object set and with objects external to the program by means of communicators such as library, external or absolute identifiers as defined in particular implementations.

Subsets may be defined but whether there are "many" is open to individual interpretation.

---

Requirement: F7

The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

---F---

Inclusion of such a capability within the language will depend upon the choice of a standard set of interfaces. Cost of this effort is even more dependent upon this choice.

---

## G. CONTROL STRUCTURES

Requirement: G1

The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

---P---

Sequential control.....T
Conditional control.....T
Iteration.....T
Recursion.....T
Parallel processing.....F
Exception handling.....P
Asyncronous interrupts.....P

The language provides the following keywords for control mechanisms:
DO
FOR
GOTO
IF
WHILE

Exception handling is provided by use of parameter labels. This does not explicitly provide all the capability required but does provide a modest start for attaining the capability.

The language does not provide an explicit mechanism for interrupt handling. Extra facilities and extensions are available to include the generation of multi-level programs, i.e., those which run under several interrupt levels.

Similarly, there is no explicit definition of parallel processing. Addition of this to the structure of the language will require major language modification.

In general, the language does not explicitly specify how any of the above requirements should be met. Parts of these requirements can be met by use of language capabilities provided for different functions --sort of twisting the intent of the language as it were.

---

Requirement: G2

The source language will provide a "go to" operation applicable to program labels within its most local scope of definition.

---P--

Go to transfers are to labels defined within the local scope of each program block. Labels are subject to the same rules as variables. Thus the "go to" statement permits jumps from an inner to an outer block or within the same block but not from an outer to an inner block.

Rules governing the "go to" operation can be modified to preclude branches out of scope but then a new mechanism for exception processing must be defined.

---

Requirement: G3

The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

---P---

Based on boolean expression.....T
Based on type from discriminated union.....F
Based on computed choice.....P

An "else" clause is not manditory for all "if-then" expressions. It may or may not be used.

A computed choice for "go to" is provided by the "switch" function. No mention is made of what happens if the switch value is outside the range of statement numbers.

There are no simple mechanisms for common cases.

---

Requirement: G4

The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g. fixed number of iteration or elements of an array exhausted).

---P---

Termination anywhere in loop.....T
Local control variables.....F
Entry at loop head only.....T
Efficient and clear simple cases.....T
Control value available at termination.....F

Termination within a loop is made possible by several constructs within the language.

Syntax rules do not explicitly state whether loops are entered only at the top.

The control variable may or may not occur within the controlled statement. The controlled variable is a word reference, i.e., either an anonymous reference or a declared word reference. The value of the controlled variable is available for declared word references. Syntax rules explicitly define the order for incrementation.

Syntax for iterative evaluation can be redone to completely meet "TINMAN" requirements.

---

Requirement: G5

Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

---P---

Recursive and nonrecursive routines.....T
Explicitly specify recursive routines.....T
No nested recursive procedures.....U

Language syntax does not state whether procedures may be defined within the body of recursive procedures. If so it is an easy task to restructure the language to prohibit this capability.

---

Requirement: G6

The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

---F---

The language does not explicitly address parallel processing requirements.

---

Requirement: G7

The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

---F---

CORAL66 does not explicitly include constructs for exception control handling. The only mechanism provided by the language is the capability to pass addresses, as LABEL parameters, for what routine to enable in the event of an exception.

Addition of exception control constructs to the language

will be a costly process.

---

Requirement: G8

There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

---P---

This requirement appears to expand on the G6 requirements. Parallel processing capabilities can be added to the language. This addition will be costly and may tend to cloud the readability and clarity of the language.

---

## H. SYNTAX AND COMMENT CONVENTIONS

Requirement: H1

The source language will be free format with an explicit statement separator, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

---P---

Free format with statement separator.....T
Mnemonically significant identifiers.....T
Conventional forms.....I
Simple uniform grammar.....T
No special case notations.....P
No abbreviation of keywords or identifiers....T
Unambiguous grammar.....P

Square brackets as used with pointers and switch functions tended to confuse the evaluator occasionally. Also, the differentiation between the uses of square and round brackets was occasionally misleading. These however, are very minor points.

---

Requirement: H2

The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedence.

---T---

The syntax is completely fixed.

---

Requirement: H3

The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

---T---

The language satisfies this requirement.

---

Requirement: H4

The language definition will provide the formation rules for identifiers and literals. These will include

literals for numbers and character strings and a break character for use internal to identifiers and literals.

---P---

Literals for numbers.....T
Character strings.....T
Break character.....F

The language does not contain a break character but can be modified to include one.

---

Requirement: H5

There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

---P---

Multiple lines are permitted within the language structure. It should be very simple to change this feature.

---

Requirement: H6

Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

---T---

The language defines 42 key words which seems to be few enough.

---

Requirement: H7

The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

---P---

Single comment convention.....F
Distinguishable from code.....P
Introduced by one or two characters.....F
Contain any charcter.....P
Appear anywhere reasonable.....T
Terminate at end of line.....F
Not prohibit reformatting.....T

Comments may be expressed in two forms; either preceded by the word COMMENT or enclosed in round brackets following a program statement. This can be easily changed to

meet the requirement.

As noted above, more than two letters are required to introduce a comment. Additionally, multiple line comments tend to confuse the reader of complex programs written in the language. These faults can be easily eliminated by modifying the language to conform to "TINMAN" requirements.

---

Requirement: H8

The language will not permit unmatched parentheses of any kind.

---T---

---

Requirement: H9

There will be a uniform referent notation.

---T---

The language employs square brackets in referring to members of arrays or tables and round brackets in defining procedure parameters. The complete intent of this requirement may not be met, however, we feel that this is a minor variation.

---

Requirement: H10

No language defined symbols appearing in the same context will have essentially different meanings.

---T---

---

## I. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

Requirement: I1

There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

---T---

This requirement is satisfied by the language.

---

Requirement: I2

Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

---P---

In general, the language does not permit defaults. However, there are some exceptions. For example, recursion is treated as a procedure for compilers lacking the recursion feature. Floating point is treated as integer for compilers not including floating point capability. In all cases of this sort the programmer is advised of defaults.

---

Requirement: I3

The user will be able to associate compile-time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

---F---

The compiler language does not support explicit specification of the object machine. These capabilities can be added to the language, but at some expense. There is a definite need for an explicit method to specify word sizes, number of bits per word, bit sizes for address formulas, and other machine dependent features.

---

Requirement: I4

The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile-time variables. In such cases, the conditional will be evaluated at compile-time and only the selected path will be compiled.

---p---

Case statements are not used for conditional evaluation. There are certain procedures available to structure the program to the object machine. For example, floating point operations are defined as integer for those machines with no floating point hardware. Recursive procedures default to procedures on those machines not having the recursion package available. Other options may be available but are not described in the evaluation documentation.

All possible options are compiled. There is no capability to select options for compilation.

---

Requirement: I5

The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

---T---

The language does indeed provide a simple, clearly identifiable base for use. Currently, the language can be easily extended to increase possible usage. As it now exists, such facilities as bit manipulation, overlay features, floating point processing, recursion, table facility, and fixed number processing can be added or deleted from the language, depending upon the object machine environment.

---

Requirement: I6

Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

---p---

Available documentation describing compiler limits is precise in some areas and vague in others. For example array dimensions and identifier lengths are explicitly defined. Nesting limits and number of identifiers were not.

The language is designed for use on a number of quite different machines. Limits are probably dependent upon the machine environment in at least some cases.

---

Requirement: I7

Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

---T---

The language satisfies this requirement.

---

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

Requirement: J1

The language and its translators will not impose run-time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

---P---

The objective of the language development has been to permit latitude, not in details but in the presence or absence of major features, which may or may not be considered worth having by the particular installation. A full compiler can provide all features but subsets are available. For example, a floating point feature would not be considered useful to an installation lacking floating point hardware. Therefore, this feature can be eliminated.

Careful use of the language will produce code which provides almost a one- for-one correspondence with machine code.

---

Requirement: J2

Any optimizations performed by the translator will not change the effect of the program.

---T---

In so far as we know, translators of the language do not change the effect of the program.

---

Requirement: J3

The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

---T---

The CODE construct provides this capability.

---

Requirement: J4

It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

---p---

Specify object presentation.....T
Specify time/space tradeoff.....F

This capability is somewhat available through use of the TABLE construct. Field orders, field widths, bit patterns, and field structures can all be specified within a table structure.

There is no mechanism for explicitly specifying tradeoffs. However, the translators do, in fact, optimize data manipulations for the object computer.

---

Requirement: J5

The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and closed routine of the same description will have identical semantics.

---p---

The open implementation for machine language insertions is clearly met by the language. Procedures are always defined to be closed. There is a macro expansion capability defined within the language, but it does not completely meet the specifications.

---

## CORAL66 EVALUATION SUMMARY

This section presents a summary of the findings of the evaluation of the CORAL66 language. Strengths and weaknesses of the language are presented. Requirements for language modification necessary to meet DOD needs and functions are addressed.

CORAL66 has a set of strengths that specifically meet the requirements of the "TINMAN". These include:

Well defined syntax -- Syntax of the language cannot be changed - Few key words are required to define the language base - No word abbreviations are permitted - Language statements are free format.

Strongly typed -- Identifiers must be fully declared before use.

Block structured -- Scope of identifiers is clearly discernable - Rules for transfers among scopes are well defined and clear.

Feature expandability -- Inclusion of various levels of major features depending upon facility configuration.

Separate compile capability -- Programs may be compiled separately and linked together for execution at run-time.

Machine dependencies excluded from language -- Base language not oriented toward any one or group of machines.

Few defaults permitted -- Defaults limited to capabilities included within specific compiler.

The major weakness identified during the evaluation was the absence of language capability. Basically, CORAL66 supports a very limited subset of DOD "TINMAN" requirements.

Limited control structure capability -- Exception handling capability very limited - Parallel processing not explicitly supported - Interrupt handling not explicitly defined within language.

Small subset of operations possible -- No provision for whole array operations - I/O constructs not available - Power sets not included - Limited Boolean capability.

Data type definition not supported -- No capability within the language to specify new data types.

Evaluation of the language did not identify any capabilities that exceeded "TINMAN" requirements. There were no

special operators or constructs that would provide a programmer with capabilities outside the scope of DOD requirements.

Syntax of the language did conform to the rules specified for the evaluation, i.e., the language syntax would not have to be modified to eliminate clashes with specifications presented within the "TINMAN".

In general, CORAL66, like many other languages of its type, provides a limited number of constructs which can be used to solve several classes of applications. The language contained no unique approaches to problem solving. Rather, the syntax follows the standard, well tested methods. The language is free of machine dependent operations and can be transported to many different machines. As is the case with other languages, specific application programs are implementation dependent and may require many modifications to operate properly on different machines.

Programs written in the language, in many cases, tend to be difficult to follow, due to the comment conventions employed. Maintainability is one of the major DOD goals.

Modification of CORAL66 to more fully support DOD requirements is possible. Some features, such as floating point capability, division with remainder, "NOT" and "NOR" operators, uniform comment notation, and typed pointer key word and syntax for example, can be easily included within the language at relatively little expense. Other features, such as expanded array and record capabilities, user type definition, I/O device definitions, power sets of enumeration, and expanded exception handling for example, will prove to be much more costly, both in terms of time and money.

# EVALUATION OF

# PASCAL

Prepared by

RLG Associates, Inc.
11250 Roger Bacon Drive
Reston, Virginia 22090

November 18, 1976

(6)

This report presents an evaluation of the PASCAL language with the requirements listed in the "TINMAN". (DOD Requirements For High Order Computer Programming Languages, "TINMAN" - 1 March, 1976, Section IV). For purposes of comparison PASCAL is considered to be defined by:

The Programming Language Pascal

There are 78 language requirements listed in Section 4 of the "TINMAN". This report compares PASCAL with each individual requirement. A summary of the degree to which the language satisfies each requirement is presented.

The introductory paragraph of each "TINMAN" requirement is included as the leading section for each requirement evaluation.

Symbols placed beside each individual requirement indicate the degree to which the language meets the requirement. The symbols and their meanings are as follows:

T - Totally meets requirement

P - Partially meets requirement

F - Does not meet requirement

U - Unknown from available documentation

A summary of the evaluation is included as the last section of this report. Merits and failures of the language as it currently is defined are discussed. Also discussed are the potential language modifications that can be made to support DOD "TINMAN" requirements.

'Requirement: A1

The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile-time and unalterable at run-time. The language will require that the type of each variable, and component of composite data structures be explicitly specified in the source programs.

---T

Every variable occurring in a statement must be introduced by a variable declaration which associates an identifier and a data type with that variable. The data type explicitly or implicitly defines the set of values which may be assumed by that variable. A data type in PASCAL may be either directly described in the variable declaration, or it may be described by a type identifier. In the latter case the type identifier must be described by an explicit type declaration.

---

Requirement: A2

The language will provide data types for integer, real (floating point and fixed-point), Boolean, and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type).

---P

| | |
|---|---|
| Integer | ---T |
| Floating point | ---T |
| Fixed point | ---F |
| Boolean | ---T |
| Character | ---T |
| Arrays | ---T |
| Records | ---T |

The values of integers are the integers within a range which are implementation dependent.

The values of floating point real quantities are a subset of real

numbers which are implementation dependent.

The values of Boolean variables are denoted by the keywords "true" and "false".

There are two character types: char and alfa. The set of values of the type char is the character set available on the printers of a particular installation. Alfa type values consist of sequences of characters whose length is defined as being implementation dependent and normally infers the number of characters packed per word on the target machine. Individual characters are not directly accessible, but alfa quantities can be unpacked from an alfa variable to a char array by a standard procedure.

Both char and alfa quantities are denoted by themselves enclosed within quotes.

In an array structure, all components are of the same type. A component is selected by an array selector, or computable index, whose type is indicated in the array type definition and which must be scalar. It is usually a programmer defined scalar type, or a subrange of the type integer.

In a record structure, the components (called fields) are not necessarily of the same type. In order that the type of a selected component be evident from the source program text at compile-time, a record selector does not contain a computable value, but consists of an identifier uniquely denoting the component to be selected. These component identifiers are defined in the record definition.

Fixed-point data types are not constituents of PASCAL nor can they be constructed from the primitive data types unless a subrange of the type real, that is with specified upper and lower bounds, is considered valid. The addition of fixed-point data types to PASCAL as either a primitive data type or as a subset of the real data type by incorporating a precision clause within the subrange declaration would appear to be a straightforward extension to the language.

---

Requirement: A3

The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. The specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

---F

PASCAL does not support the concept of precision specification of floating point variables either at a block structure level or on an individual variable basis. It would be a straightforward proposition to extend the language to require that each block in which floating point variables are declared include a declaration of the precision required for arithmetic operations performed within that block and for the actual data representations of local floating point variables. However, it is anticipated that to allow unique precision specifications for each variable would result in possibly inefficient object code and an excessive compile-time burden resulting from increased subtype checking and the inclusion of object code for data representation conversion. Alternatively the precision could be utilized by the compiler only to determine if the target machine were capable of supporting the specified precision.

---

Requirement: A4

Fixed-point numbers will be treated as exact quantities which have a range and fractional step size which are determined by the user at compile-time. Scale factor management will be done by the compiler.

---N/A

Since PASCAL does not include a real fixed-point data type, either as a primitive or the ability to generate it from its primitives, then the question of fixed-point range and resolution are not applicable. However, if the extension concerning the introduction of real fixed-point data types described in the response to Section A2 were implemented as a primitive data type, then the resolution and lower and upper bounds could be specified as an extension to the subrange clause. The information within such a clause would provide all the attributes necessary for the compiler to handle scale management and produce object code for bound checking, if desired.

---

Requirement: A5

Character sets will be treated as any other enumeration type.

---P+

The internal representation of PASCAL char and alfa type data items is not defined within the language and would most probably be fixed for any given implementation. However, the ability to define scalar types will allow any desired coallation sequence for an alphabet. As a result several alphabets, including ASCII and EBCDIC, could be defined, thus, mapping from one to another would be a trivial task. The ASCII and EBCDIC alphabets are not predefined within the language specification, thus, requiring each user to declare them according to the definable scalar format.

---

Requirement: A6

The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array. The number of dimensions, the type of lower subscript bound will be determinable at compile-time. The upper subscript bound will be determinable at entry to the array allocation scope.

---P+

An array type is a structure consisting of a fixed number of components which are all of the same type, referred to as the component type. The elements of the array are designated by indices, values belonging to the so-called index type. The array type definition specifies both the component type and the index type. Both lower and upper bounds are normally specified at compile-time. However, utilization of the class type definition in conjunction with standard procedure allocation provides the ability to allocate the array at scope entry. This is done dynamically, which is in conflict with the "TINMAN" requirement but the class type definition requires that the maximum array size be explicitly specified at compile-time. It is our opinion that this requirement satisfies the intent of the "TINMAN" requirement.

---

Requirement: A7

The language will permit records to have alternative structures, each of which is fixed at compile-time. The name and type of each record component will be specified by the user at compile-time.

---P+

A record is a structure consisting of a fixed number of components, possibly of different types. The record type definition specifies for each component, called a field, its type and the identifier which denotes it. A record type may have one or more alternative structures, referred to as variants, as specified by the "TINMAN". However, at a given time, the specific variant is selected by the value of a single fixed field of the record, referred to as the tag field. This is in conflict with the "TINMAN" requirement which requires that alternative structures be selected by a general Boolean expression. The extension of the language to satisfy this subrequirement is considered to be extensive since it would imply a totally different selection mechanism than the currently specified tag field.

Requirement: B1

Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or a union type containing that type. Reference will retrieve the last assigned value.

---P+

The PASCAL assignment statement will allow a variable of any given type, with the exception of class and file types to be assigned to an expression of the identical type. Thus, an entire array can be equated to an equivalent array, an entire record to an equivalent record, or a component of a record to a similar equivalent component or primitive variable of identical type. There are two relaxations to this general rule of type consistency:

1) Where the type of the dependent variable is real and the type of the expression is integer or a subrange thereof.

2) Where the type of the expression is a subrange of the independent variable.

These type checking relaxations could readily be deleted from any given implementation, however, the ability to assign file and class type data items would be considered a major impact and would require a detailed analysis to determine any undesirable side effects or conflicts.

---

Requirement: B2

The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

---P+

The PASCAL language identity operators for equality (=) and inequality (≠) may be used with operands of any type with the exception of class and file types, the result is always type Boolean.

To include the operand types class and file would be considered a major impact for the same reasons as described in the response to Section B1 of this report.

---

Requirement: B3

Relational operations will be automatically defined for numeric data and all types defined by enumeration.

---P+

The relational operators defined for PASCAL are as follows:

| | |
|---|---|
| = | ≠ |
| < | > |
| ≤ | ≥ |

The identity operators (=,≠) may be used with any data type with the exceptions of those noted in Section B2. The relative operators (<,>,≤,≥) may be utilized with any scalar or subrange type. The scalar types include the predefined items, integer, real, Boolean, char and alfa, and those defined by enumeration. It should be noted here that the semantics of the relative operators within the alfa context are not obvious and are defined below.

If X and Y are alfa identifiers, the value of which consist of sequences of n characters, where n is an implementation dependent parameter, (see Section A2) then:

X<Y if Xi=Yi for 1<i<k-1 and Xk<Yk

X>Y if Xi=Yi for 1<i<k-1 and Xk>Yk

---

Requirement: B4

The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed-point arguments and remainder), and negation.

---P+

The arithmetic operators of PASCAL are defined as follows:

| | |
|---|---|
| * | multiplication |
| / | division |
| div | division with truncation |
| mod | m mod n=m-((m div n)*n) |
| + | addition or identity |
| - | subtraction or inversion |

The multiplication (*), division (/), addition (+), and subtraction (-) operators permit the operands to be real, integer, or mixed. The result for multiplication, addition, and subtraction is integer only if both operands are integer, else it is real. The result for the division operator is always real. The div and mod operators may be utilized with integer operands only, yielding integer results of a truncated integer division and remainder, respectively. There is no exponentiation operator. To add that feature would be relatively minor if a decision concerning the semantics and result type could be resolved. For example, the expression A**X**Y if evaluated with a conventional left to right precedence is equivalent to A**(X+Y) which some would argue, is not what was intended. If evaluated with a right to left precendence, the result is equivalent to A***(X**Y). This results in the relatively complex semantic rules that in the absence of parenthesis expression operators are evaluated according to precedence and left to right with the exception of the exponentiation operator which is evaluated right to left.

In addition, the result type of an exponentiation operation, when both operands are integer also causes considerable confusion unless clearly defined. Finally, the possibility of complex results introduces a further confusion factor. The designer(s)

of PASCAL were perhaps wise in omitting the exponentiation operator from the arithmetic operator set.

---

Requirement: B5

Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will be on the least significant digits and will never be implicit for integers and fixed-point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

---P

No truncation of most significant digits ---U

No implicit truncation or rounding for fixed-point and integer operands ---T

Implicit rounding beyond specified precision ---N/A

The source material used for this report was addressed to the PASCAL language definition, not a compiler implementation, thus, it was not possible to determine if most significant digit truncation would ever occur. Stating that it should never occur is an incomplete specification, since there are obvious instances where it could occur, the most common, perhaps, is the assignment of a floating point expression to an integer or fixed-point operand. Since the latter will generally have a more restricted range then should that range be exceeded what action is to be taken? In most environments the choice is to either ignore the discrepancy resulting in what is essentially modulus arithmetic or to abort the execution of the program and return control to the system executive. It would certainly be feasible and, perhaps, desirable to the HOL project to provide the source language with a mechanism to specify those actions such that the user would retain control, if desired. This could be accomplished by optionally specifying at the entrance to each block an identifying procedure label which would be executed if overflow occurs. Within that procedure, the user would determine the need to either abort program execution or to continue.

Implicit truncation cannot occur within PASCAL assignments since a real expression cannot be assigned to an integer operand. In addition, within an expression, mixed mode operations always result in a real value. Explicit truncation of real expressions can be achieved by use of the standard function, trunc.

Since floating point precision is not an attribute of the

'language, implicit rounding beyond a specified precision is not applicable. However, if precision specification were included as described in Section A3 then it is anticipated that a real expression would be evaluated within the maximum precision supported by the target machine, only the operation of assignment would round the result to the specified precision.

---

Requirement: B6

The built-in Boolean operations will include "AND", "OR", "NOT" and "NOR". The operations on scalars will be evaluated in short circuit mode.

---P

The built-in Boolean operations of PASCAL are as follows:

$\wedge$(logical "AND")

$\vee$(logical "OR")
$\neg$(logical "NOT")

The operation "NOR" is not explicitly available within the language. It is anticipated that it could be incorporated with a minimum effort. However, purists may argue that it is not only unnecessary but undesirable since both "NOR" and "AND" can be represented by use of the "NOT" operator applied to a subexpression in parentheses.

The language definition does not specify the semantics of short-circuit evaluation, however, any implementation could of course, employ short-circuit evaluation techniques.

---

Requirement: B7

The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

---T

Scalar operations on arrays ---T(U)

'Assignment between records and arrays of comformable type ---T

It is clear from the PASCAL language definition that assignments between records and arrays of conformable type are valid. In addition, entire variables representing arrays can apparently be operands of expressions. However, the semantics of such operations is undefined, thus, multiplication of arrays could imply matrix multiplication. The obvious solution to this would be to include explicit semantic definitions within the language specifications.

---

Requirement: B10

The base language will provide operations allowing programs to interact with files, channels, or devices (including terminals). These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

---P+

A file structure in PASCAL is a sequence of components of the same type. A natural ordering of the components is defined through the sequence. At any instant, only one component is directly accessible. The other components are made accessible through the standard file positioning procedures: put, get and reset. At any given time, a file is in one of three modes called input, output and neutral. According to the mode, a file can be read sequentially or it can be written to by appending components to the existing sequence of components. File positioning procedures may determine the current mode. The file type definition does not determine the number of components, and this number is variable during the execution of the program. The components of a file may be a simple primitive data type (i.e., integer) or may be complex structures, the specific form of the structures being defined by a record definition. Two standard file variables can be considered as predeclared for all implementations as:

```
input, output: file of char
```

The file input is restricted to input mode (reading only), and the file output is restricted to output mode (writing only). A PASCAL program should be regarded as a procedure with these two variables as formal parameters. The corresponding actual parameters are expected either to be standard input and output devices of the target computer installation or to be assignable by the system commands of the target computer. As a result, it would appear that files would not normally be dynamically assignable, however, additional standard procedures could be added to a given

implementation to provide a dynamic reassignment capability.

The standard file positioning procedures are defined below:

put (F) Advances the file pointer of file F to the next file component. It is only applicable if the file is in either the output or neutral mode. After execution, the file is in the output mode.

get (F) Advances the file pointer of file F to the next file component. It is only applicable if the file is in either the input or neutral mode. If there does not exist a next file component, the end of file condition arises, the value of the variable denoted by the file pointer becomes undefined, and the file is put into the neutral mode.

reset (F) The file pointer of file F is reset to its beginning, and the file is put into the neutral mode.

N.B. Initially a file variable is in the neutral mode.

In addition to the file positioning procedures, a further standard function is available to determine the end-of-file status of a file. The function is eof with the file variable as an argument. The function type is Boolean which when true, indicates an end-of-file status. Any other conditions, such as data transfer error, are not available as standard functions but could be included in any given implementation at an anticipated low to moderate cost.

---

Requirement: B11

The language will provide operations on data types as power sets of enumeration types (Section E6). The operations will include union intersection, difference, complement, and an element predicate.

---p

A PASCAL powerset type defines a range of values as the powerset of another scalar type, the so-called base type operators applicable to all power set types are:

union

* intersection

- set difference

''' membership

----------p

Requirement: C1

Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left to right.

---T

The source documentation specifies that sequences of operators of the same precedence level are executed from left to right. The rules of precedence are reflected by the specified syntax definition.

---

Requirement: C2

Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

---T

Expressions are constructs denoting rules for obtaining values of variables and generating new values by the application of operators. Expressions consist of operands (i.e., variables, constants and functions) and operators.

The rules of composition specify operator precedences according to four classes of operators. The Boolean negation operator () has the highest, multiplying operators (*,/,div,mod,^), then the so-called adding operators (+,-,V), and finally, with the lowest precedence, the relational operators (=,≠,<,>,≤,≥). These precedence levels are fixed and cannot be altered by the user as required by the "TINMAN". In addition, conventional explicit parenthetical pairs can be included within expressions, to any desired level, to specify the intended evaluation sequence or to ensure the required evaluation order of operators of the same precedence level.

---

Requirement: C3

Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

---T

---

Requirement: C4

Constant expressions will be allowed in programs anywhere constants are allowed, and constants will be evaluated before runtime.

---F

PASCAL does not support the concept of constant compile-time expressions either in the form of literal or parameterized constants. Parameterized constants are definable and may appear at any point where a literal constant is specified. Parameterized constants are generally favored, since they allow a single assignment of a compile-time identifier to a value which may subsequently be utilized by numerous references. Coupled with the capability of compile-time expression evaluation, compile-time identifiers may be functions of other similar identifiers, thus minimizing programmer errors and providing a readily maintainable environment. The addition of such a capability to PASCAL would be considered a moderate effort.

---

Requirement: C5

There will be a consistent set of rules applicable to all parameters, whether they be for procedures, types, exception handling, parallel processes, declaration, or built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters.

---T

Consistency has been achieved for parameter handling within PASCAL, primiarly inherited from its predecessor ALGOL 60. Identifier references, be they actual or formal, are fully typed to ensure that consistency.

---

Requirement: C6

Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at

compile-time. The size and subscript range for array parameters need not be determinable at compile-time but can be passed as a part of the parameter.

---P

Actual and formal parameters will agree in type.
---T

Rank of parameter arrays is fixed at compile-time.
---F

Parameter array upper and lower bounds can be passed.
---T

Actual parameters are, of course, defined by the normal data declaration constructs. Formal parameters are declared, or more correctly specified, by appending the appropriate clause to the formal parameter synonym. There exist four clauses to specify the four kinds of parameters: variable, constant, procedure, and function.

In the case of variable-parameters the actual parameter must be a variable. If it is a variable denoting a component of a structured variable, the selector is evaluated prior to execution of the procedure. If the parameter is a constant parameter, the corresponding actual parameter must be an expression. If the parameter is a procedure or function, the corresponding actual parameter must be a procedure or function identifier, respectively. Actual to formal parameter correspondence is achieved by the ordinal positions of the parameters in the lists of actual and formal parameters and bounds.

The rank of arrays are not specified by the formal parameter declarations and, thus, must be passed as a part of the actual parameter upon invocation of the procedure.

---

Requirement: C7

There will be only four classes of formal parameters. For data there will be those acting as constants representing the actual parameter value at the time of call and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

---P+

Call by value parameters ---T

Call by name parameters ---T

Exception handling parameters ---U

Procedure identifier parameters ---T

The types of procedure parameters available to PASCAL are defined in Section C6 of this report. They are: constant, variable, procedure, and function. Constant and variable parameters are equivalent to "call by value" and "call by name" parameters, respectively. The procedure identifier parameter allows procedures to be passed as actual parameters to other procedures as mandated by the "TINMAN".

It is unclear from the source documentation as to the availability of exception handling parameters, most commonly encountered, in the form of labels.

---

Requirement: C8

Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run-time.

---P-

Specified attributes optional ---F

Attributes unalterable at run-time ---T

All procedure formal parameters are fully typed within the procedure heading to ensure consistency of type with the actual parameters encountered at procedure invocation. If those formal parameter attributes were optional, it is difficult to envision how type consistency could be effectively managed. In fact, this requirement appears to be in conflict with much of the "TINMAN" since overall data type consistency is one of the major conventions dictated.

---

Requirement: C9

There will be provisions for variable numbers of arguments, but

in such cases, all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile-time .

---F

PASCAL procedure parameters are limited to the fixed number specified by the declaration. Each procedure invocation must, of course, include the same number of actual parameters.

The concept of variable numbers of arguments to procedures is, thus, somewhat foreign to this language. However, within the specified restraints, it would appear to be a realistic requirement. The impact of including the capability within a given language implementation would be moderate to major.

Requirement: D1

The user will have the ability to associate constant values on any type with identifiers.

---T

The constant definition facility provided enables an identifier to be a synonym for a constant. The constant may be a signed or unsigned real, or integer number, a character string, a user defined identifier, or the undefined pointer constant nil.

---

Requirement: D2

The language will provide a syntax and a consistent interpretation for literals of built-in data types. Numeric literals will have the same value (within the specified precision) in both programs and data (input or output).

---T,N/A

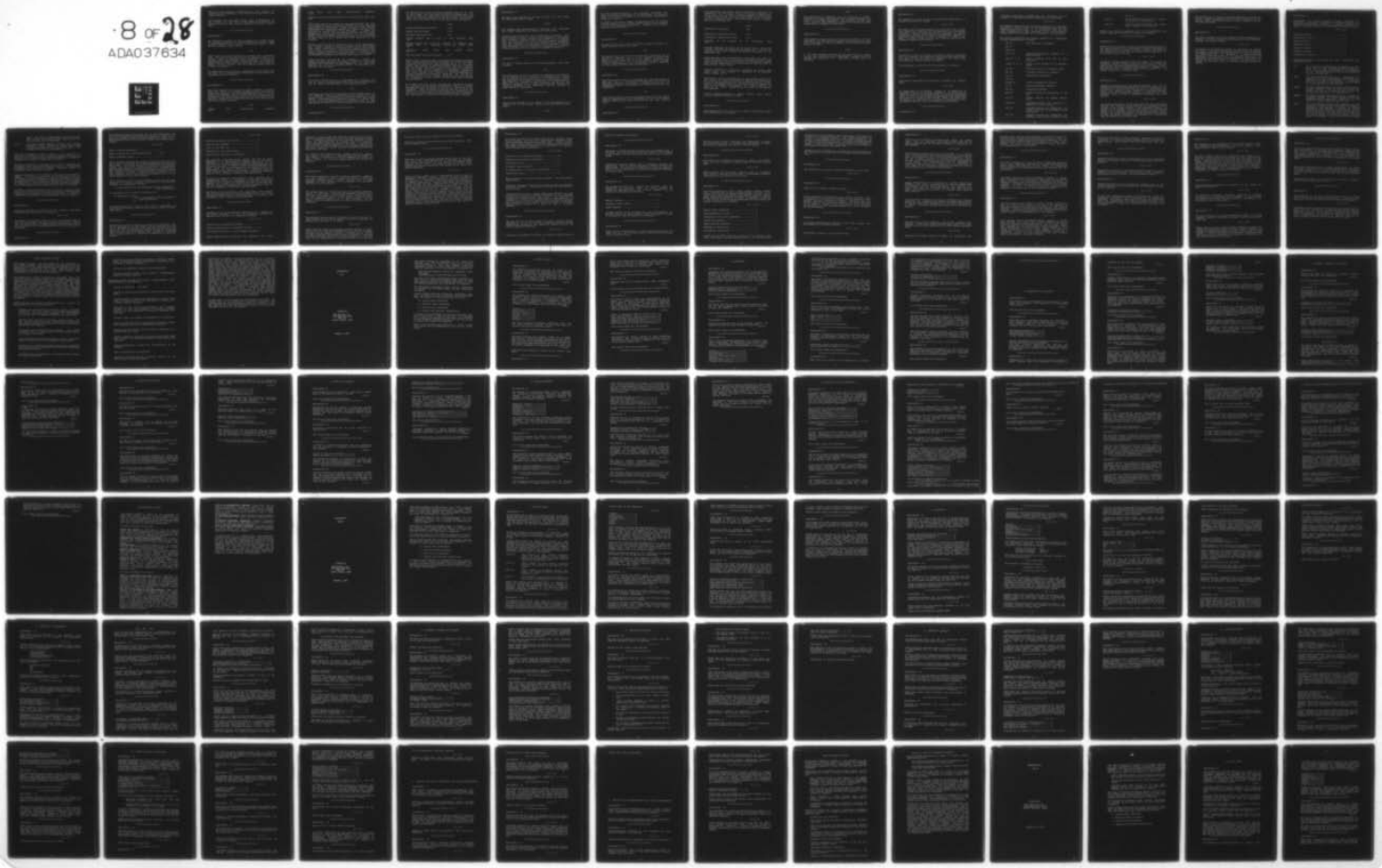
Consistent literal syntax and interpretation ---T

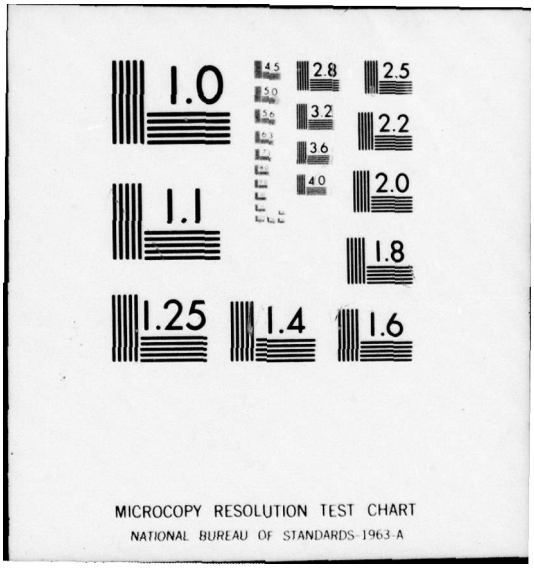
Equivalent literal values for programs and data ---N/A

AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

8 OF 28
ADA037634




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
10.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

There is a single syntactic definition and hence implied interpretation for references to all literals thus providing a consistent notation.

The requirement for equivalent literal value representation in both program and data input is considered out of scope of the language definition which does not address the run-time environment.

---

Requirement: D3

The language will permit the user to specify the initial values of individual variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

---F

PASCAL does not support the concept of variable data initialization. The value of variables of entry of allocation scope is undefined. Since, for efficient memory management it is normal for noninterfering program blocks to share data space, initialization cannot occur at either compile or load time and must become a dynamic run-time feature, this is in partial conflict with the "TINMAN" requirement.

The extensions to the language to accomodate initialization would be minor but the impact on data management to fully satisfy the requirement would be moderate.

---

Requirement: D4

The source language will require its users to specify, individually, the range of all numeric variables and the step size for fixed-point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

---F

Numeric range specification mandatory
---F

Fixed point step size specifications mandatory
---N/A

Range and step size specifications do not define a new data type
---F

PASCAL does support the concept of range specification for both the predefined scalar quantities (integer, etc.) and those data types defined by enumeration. However, contrary to the "TINMAN" requirement such range specifications are optional and each range specification is considered a unique data type. To modify the language to require that all numeric variables require a range specification would require minimal effort, but the conceptual change of not considering such subrange types to be unique would be major, since it would change the total structure of the language.

Perhaps, the object of the requirement is to allow assignments, etc., between variously ranged variables of the same general class without violating the stringent type checking rules specified within the "TINMAN". As previously described within this report, PASCAL conveniently skirts this issue by relaxing type checking between scalars and subranges thereof. If this approach is considered acceptable, the overall impact would be minimal.

Since fixed-point variables are not supported by PASCAL, the question of step size is not applicable. However, if fixed-point variables were introduced, it would appear that for total consistency and for little additional effort, the step size attribute could be introduced.

---

Requirement: D5

The range of values which can be associated with a variable, array or record component will be any built in type, any defined type or a contiguous subsequence of any enumeration type.

---T

PASCAL data declarations are recursive in that element values of all identifier types can themselves be typed identifiers. This permits arrays to be components of records or arrays and permits records to be components of records or arrays. An element value may be a contiguous subsequence of any enumeration type if that subsequence is defined as a subrange type of the entire sequence.

---

Requirement: D6

The language will provide a pointer mechanism which can be used to build data with shared and for recursive substructure. The pointer property will only affect the use of variables (including array and record components) of same data types. Pointer variables will be as safe in their use as are any other variables.

---P+

Pointer mechanism(s) available ---T

Shared data structures ---T

Recursive data structures ---T

Pointer property type is bound to data structure type ---T

Pointer property will be only for variables of composite components and of composite array and record types ---F

Allocation scope wider than access scope ---U

Type and access restricted pointers ---T

PASCAL 'class' variables which are dynamically assigned by reference to the standard procedure allocation may only be referenced by pointer type variables. Class variables may be of any type, either user defined or predefined scalar, which is in conflict with the "TINMAN" requirement which dictates that only composite data structures may be referenced by pointers. A pointer is bound to a given 'class' variable by an appropriate declaration and, thus, ensures type checking for pointer variables in a fashion compatible with nonpointer variables. The limiting of be a minor change to the language. Since more than one pointer may be bound to a single 'class' variable shared data structures can be achieved.

It is clear that since 'class' variables are dynamically allocated, recursive data structures are feasible. However, the source documentation does not specify if and how such data space is deallocated. Since allocation is explicit but must be within the bounds of the pointer scope it would appear that references could occur prior to the initial allocation unless compiler topographical flow analysis is included or run-time checks are performed.

Requirement: E1

The user of the language will be able to define new data types and operations within programs.

-----P+

The extensive user data declaration facilities fully satisfies the "TINMAN" requirement for defining new data types.

Operations can be defined, only by functions or procedures which do not provide a means of defining infix operators. The requirement to define new infix operators would appear to be in conflict with H2 which specifies that "new infix operator precedence" shall not be definable. If new infix operators are to be defined, both the semantics and precedence of those operators would require definition unless the definition were described in terms of existing operators in which case the precedence is implied.

---

Requirement: E2

The "use" of defined types will be indistinguishable from built in types.

---T

The syntactical structure of PASCAL for reference to any identifier or constant, be it a built-in or user defined type is identical thus satisfying this requirement. The semantics of a statement referencing data types is a function of the general class of data types (e.g., scalars) be they built-in or user defined. For example a relational test of scalar identifiers has both the same syntax and semantics if the identifiers represent integers or those defined by enumeration.

---

Requirement: E3

Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

---P+

The base language provides all the elementary components from which more complex components may be constructed. The constructions may exist in the program, or if the specific implementation allows, in a library.

In general, there are no default declarations within the language with the exception of the constant specificator, const, associated with formal parameter delcarations. To require this keyword would be a trivial effort.

---

Requirement: E4

The user will be able, within the source language, to extend existing operators to new data types.

---p

The operand types with which the defined set of operators can be applied is specified and it is not possible to extend those operators to additional variable types. However, since user defined variable types fall into a given set of general classes (scalars, powersets, etc.), the appropriate existing operators for the operands general class are applicable.

---

Requirement: E5

Type definitions in the source language will permit definition of both the class of data objects comprising the type and set of operations applicable to the class. A defined type will not automatically inherit the operations of the data with which it is represented.

---p

User defined variable types, as explained in E4, fall into one of several classifications. The operations pertinent to the general type are inherited and cannot be modified by the user declaration.

---

Requirement: E6

The data objects comprising a defined type will be definable by enumeration of their literal names, as cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the powerset of an enumeration type. These definitions will be processed entirely at compile-time.

---P+

Definable by enumeration ---T

Definable by cartesian products ---T

Definable by discriminated union ---F

Definable by the powerset of an enumeration type ---T

The basic definable data types are the scalar types. Their definition indicates an ordered set of values, i.e., introduces an identifier as a literal constant representing each value in the enumeration set.

Structured data types are described by describing the types of their components and by indicating a structuring method. Both array and record structure are construed as definitions by cartesian products since they provide definitions in terms of other in built or user defined types.

Powerset definition is explicitly available for scalar base types, either those defined by enumeration or the predefined types (integer, etc.).

Definition by union of disjoint sets is not included though such sets could, of course, explicitly be defined by enumeration of all of the literal elements from the desired disjoint set. Alternatively the superset could by initially defined and subsets defined by the subrange types feature. With either approach, it would be the users' responsibility to ensure that the subsets were disjoint.

To add a language construct to provide disjoint union, typing would be considered relatively minor.

---

Requirement: E7

Type definitions by free union (i.e., union of nondisjoint types) and subsetting are not desired.

---p-

Type definitions by free-union are not available in PASCAL. Subranging of scalar types is permitted and such subranges, contrary to the "TINMAN" requirement, are construed as new data types. To alter these semantics to satisfy the requirement would be considered a moderate to major task.

---

Requirement: E8

When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at time of allocation and deallocation of variable of the type.

---p-

n. The source documentation used for this report did not specify if and how allocated data is deallocated. However, it is apparently not explicit and thus finalizatin actions cannot be specified.

Requirement: F1

The language will allow the user to distinguish between scope of allocation and scope of access.

---- F ----

The PASCAL block structure dictates that the access and allocation scopes are identical with the exception of the class type variables. Since the allocation of class variables is dynamic and explicit, unless a given compiler implementation includes topographical flow analysis, it would be possible for a user to attempt to access a class variable, via a pointer, outside of the allocation scope of that variable.

---

Requirement: F2

The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

The requirement is partially met, as described in F1 above.

---P

Requirement: F3

The scope of identifiers will be wholly determined at compile-time.

---- T ----

All identifiers must be defined. Access to an identifier is guaranteed within the scope of declaration. Identifiers may be redefined either within imbedded scopes, in which case references apply to the most deeply imbedded declaration scope surrounding the reference or within disjoint scopes. These rules ensure that the scope of all identifiers is a function of the program's static structure and are, thus, determinable at compile-time.

---

Requirement: F4

A variety of application oriented data and operations will be available in libraries and easily accessible in the language.

---- P ----

The PASCAL language does not provide the mechanism for including compile-time library files in which application oriented data structures could be defined. It is anticipated that for any given implementation, data declaraton files could be readily merged either by an extension to the language or a preprocessor.

The following operations are, however, assumed to be predeclared in every implementation of PASCAL:

| | |
|---|---|
| put (f) | File positioning procedures. |
| get (f) | |
| reset (f) | |
| alloc (p) | Allocate an additional component to a class variable. |
| pack (a, i, z) | Pack a character array into an alfa variable. |
| unpack (z, a, i) | Unpack an alfa variable into a character array. |
| abs (x) | Arithmetic value for arithmetic value. |
| sqr (x) | Arithmetic function for squaring. |
| sin (x) | Trigonometric functions. |
| cos (x) | |
| arctan (x) | Inverse trigonometric function. |
| ln (x) | Natural logorithmic function. |
| exp (x) | Exponential function. |
| add (x) | Boolean function for determining x mod 2=1. |
| eof (f) | Boolean function for sensing end-of-file. |
| trunc (x) | Arithmetic function for obtaining integepart of a real quantity. |
| int (x) | Arithmetic function for determining the ordinal position of a character in its defined set. |
| chr (x) | Character function for determining the character whose ordinal position in the |

set is x.

succ (x) — Scalar function for finding the successor value of x from its set.

pred (x) — Scalar function for finding the predecessor value of x from its set.

Further the language specifies that any implementation may feature additional predeclared functions or procedures.

---

Requirement: F5

Program components defined within the current program and not in the base language will be maintained in compile-time libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

---- F ----

The PASCAL language definition does not address the concept of externally defined modules in any form, since such features have traditionally been considered functions of the compiler environment and are frequently implementation dependent. However, it is believed that such an extension could readily be appended to the PASCAL language for a moderate effort.

---

Requirement: F6

Libraries and compools will be indistinguishable. They will be capable of holding anything definable in the language and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

---- F ----

The language definition, as with Requirement F5, does not address compiler and run-time environments nor any references to items external to the current program. In the practical situation, of course, it is desirable to provide the facility to include items defined within compools and to explicitly reference library items. To ensure the integrity of type checking would require, at least for library references, the maintaining of compile-time data in addition to the object code.

The additions to the language to enable references to compool and library elements is almost trivial, however, to estimate the impact on a given implementation would be futile because of the environment dependency.

---

Requirement: F7

The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

---- P ----

In addition to the specified standard I/O procedures for accessing files, the language definition provides for an extended set of standard procedures. This feature could be readily utilized the capabilities for this requirement. However, it would be desirable to include, as part of the language definition, the identifiers and protocol of those procedures to ensure that standardization could be achieved. The effort to do this cannot, of course, be estimated until the specific functions required are defined.

Requirement: G1

The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structure for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling.

---- P ----

Sequential control . . . . . T

Conditional control . . . . . T

Iterative control . . . . . T

Recursive control . . . . . T

Parallel processing . . . . . F

Exception handling . . . . . F

Asynchronous interrupt . . . . . F

The language elements which provide the above requirements are identified below:

IF: The IF statement specifies that a statement be executed only if the specified Boolean expression is true. If it is false, no statement is to be executed or the alternate statement following the connective ELSE is to be executed.

CASE: The CASE statement consists of an expression (the selector) and a list of statements, each labeled by a constant of the type of the selector. It specifies that one statement be executed whose label is equal to the current value of the selector.

WHILE: The WHILE statement specifies that a statement be repeatedly executed while the value of a Boolean expression is true. If its value is false at the beginning, the statement is not executed at all.

REPEAT: The REPEAT statement specifies that the sequence of statements between the symbols REPEAT and UNTIL are repeatedly (and at least once) executed until the value of a Boolean expression becomes true.

FOR: The FOR statement indicates that a statement is to be repeatedly executed while a progression of values is assigned to a variable which is called the control variable for that statement. Expressions are included to specify the initial and final values for the control variable. At each iteration, the value of the control variable is replaced by either the successor or predecessor of the current control variable

value. The choice is determined by use of the alternate symbols TO or DOWN TO which separates the initial and final value expressions.

GO TO: The GO TO statement serves to indicate that further processing should continue at another part of the program text, namely at the place of the label. The transfer of control is unconditional.

The scope of access of a label is limited to the structure in which it is defined, and all structures defined within it, with the exception of procedure and function defintions. In those instances, it is invalid to branch out of or into.

Recursive structures are available in the form of procedure declarations where only direct recursion is valid. This property is implied by a reference to a procedure identifier within the executable test of the same procedure.

Parallel processing is not explicitly available as a structure of PASCAL. If it were to be included on a limited basis by the introduction of definable tasks, in a fashion syntactically similar to procedures and if identifier references were limited to local declarations and formal parameters, the effort would be moderate. If capabilities beyond this were required, extensive consideration of the protocol of data handling would need to be defined.

Similarly, interrupt handling is not explicitly defined though they could be processed by the inclusion of further standard procedures, and such a method may prove satisfactory if agreement concerning the semantics of such procedures could be reached.

---

Requirement: G2

The source language will provide a "GO TO" operation applicable within its most local scope of definition.

---- P ----

The PASCAL GO TO statement does not limit the referenced label to the most local scope. The labels obey the rules of all identifiers, in that the scope of access includes the structure in which it is defined and all structures declared within the structure of definition.

---

Requirement: G3

The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

---- P ----

Based on Boolean expression . . . . . T

Based on subtype from discriminated union . . . . T/U

Based on computed choice . . . . . T

The IF statement provides the primary control structure and dictates that the condition be a Boolean expression as required by the "TINMAN". Data types defined by enumeraton and as powersets thereof may utilize the IN operator which may be construed as a discriminated union but returns a Boolean value. The CASE statement provides conditional statement execution based on the value of a variable of any type. No explicit action is defined for a CASE statement should the value of the expression be out of the bounds specified by the CASE statement.

The IF statement is not fully partitioned, in that the ELSE substructure is optional. However, the semantics of the following statement are unambiguously defined:

```
IF <expression-1> THEN IF <expression-2> THEN <statement-1>
                                         ELSE <statement-2>
```

by interpreting the construct as equivalent to:

```
IF <expression-1> THEN
                  BEGIN IF <expression-2> THEN <statement-1>
                        ELSE <statement-2>
                  END
```

It would appear that the intent of the "TINMAN" requirement is met though it requires a semantic definition to qualify the ambiguous syntax.

---

Requirement: G4

The iterative control structure will permit the termination condition anywhere in the loop, require control variables to be local to the iterative control, allow entry only at the head of the loop, and not impose excessive overhead in clarity or run-time execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

---- P+ ----

Termination anywhere within loop . . . . . P

Local control variables . . . . . F

Entry at loop head only . . . . . T

Efficient and clear simple cases . . . . . T

Control value available at termination . . . . . U

The iterative structures WHILE, REPEAT, and FOR are fully described in G1 and satisfy the general intent of the "TINMAN" requirement. The WHILE statement provides termination at the head of a loop while the REPEAT statement provides termination at the end of the loop. In addition, exit may be achieved from any point within the loop by combining an IF statement with the unconditional GO TO. However, use of such a technique does not limit program execution to the statement following the iterative structure as is, perhaps, intended by this requirement.

The control variable of a FOR statement, which provides for a predetermined number of iterations, is not local to iterative structure and its value at termination is undefined, unless termination is a result of a branch (GO TO) out of the structure. The only restriction concerning the control variable is that it may not be assigned a value within the structure.

The impact of fully satisfying this requirement would require careful consideration of the total language, since it may violate the general rules concerning identifier scope. As a result, the anticipated effort involved was not predicted.

---

Requirement: G5

Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

---- P ----

Recursive and nonrecursive routines . . . . . T

Explicit specification of recursive routines . . . F

No procedure definitions within recursive routines F

PASCAL procedures may be recursive, but apparently only direct

recursion is valid, since that property is implied only be reference to a procedure identifier within the same procedure declaration. Recursive procedures are not explicitly declared. In addition, all identifiers must be declared prior to reference, thus, precluding indirect recursion at any level. A further violation of the "TINMAN" requirement is that procedure declarations are valid within more global procedure declarations be they recursive or otherwise.

As a result of the somewhat unclear language definition concerning indirect recursion and the uncertainty of the "TINMAN" on this matter, it is difficult to make an assessment on overall impact. However, it is safe to assume that at best it would be moderate.

---

Requirement: G6

The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

---- F ----

The language does not explicitly provide for parallel processing, pseudo or otherwise. This item was discussed in Section G1 of this report and, although any extension of the language to satisfy this requirement would be a major impact, it could be minimized if constraints described in G1 were provided. The net result of such an extension would only partially satisfy this requirement.

---

Requirement: G7

The exception handling control structure will permit the user to cause transfer and control of data for any error or exception situation which may occur in a program.

---- P ----

PASCAL does not specifically address exception handling as there are no control constructs dedicated to that function. However, extensions to the nebulous 'standard' procedures and the ability to pass procedure identifiers as actual parameters to procedure invocations provides an exception handling capability. The protocol of this requirement is, thus, not dictated by the language

and would require rigorous definition for a given system.

Addition of formally defined exception handling constructs would be an extensive effort.

---

Requirement: G8

There will be source language features permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, give access to real time clocks, permit asynchronous hardware interrupts to be treated as any other exception situation.

---- F ----

This requirement appears to be a consolidation of G1, G6, and G7 addressing interrupt handling, parallel processing, exception handling, etc. As previously described within those sections, it is feasible that they could be partially satisfied by extensions of existing capabilities. However, since PASCAL does not specifically address real time processing it is probable that the extensions to satisfy these requirements should take the form of new constructs to the language. It would be necessary to undertake a detailed analysis of the language to ensure that the ground rules of the current language are not violated nor are seriously overlapping functions created. These criteria may seem obvious but an examinatton of other languages reveals that inconsistencies, incomplete definitions, and functional overlap are all too often encountered. The impact on the language to satisfy this requirement would, no doubt, be major but without a detailed analysis, which is considered beyond the scope of this report, an accurate estimate cannot be made.

Requirement: H1

The source language will be free format with an explicit statement separator, allow the use of mnemonically significant identifiers, be based on conventional forms, have a simple uniform and easily parsed grammar, not provide unique notations for special cases, not permit abbreviation of identifiers or keywords, and be syntactically unambiguous.

---- P+ ----

Free format with statement separator . . . . . P+

Mnemonically significant identifiers . . . . . T

Conventional forms . . . . . T

No special case notations . . . . . T

No abbreviations of keywords or identifiers . . . T

Unambiguous grammar . . . . . P+

This requirement is almost entirely satisfied. The discrepancies appear below:

The source language is totally free format and does not require a statement separator. This feature is achieved by the simplicity of the syntax.

There appears to be only a single syntactical ambiguity, to this evaluator, concerning the optional ELSE clause of an IF statement as described in Section G3 of this report. The ambiguity is resolved by a precise semantic definition. Alternatively the deficiency could be solved by defining the ELSE clause as mandatory and introducing a null statement into the language. This latter feature would be required as a result of lack of statement separator.

---

Requirement: H2

The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierachies, introduce new precedence rules, define new keyword forms, or define new operator precedence.

---- T ----

The syntax and semantics of PASCAL are completely defined and can

neither be altered nor extended.

---

Requirement: H3

The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 ASCII character subset.

---- P ----

Although the required character set is considered suitable for publication, the 89 unique characters preclude the use of an ASCII subset. Reduction of the character set could be reduced to conform to that requirement by restricting alphabetic characters to a single case.

---

Requirement: H4

The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

---- P+ ----

Numeric literals . . . . . T

Character string literals . . . . . T

Break character . . . . . F

A break character is not included within the valid character set but could readily be added, however, since the language is totally free form it does not appear to be necessary.

---

Requirement: H5

There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

---- F ----

The free format of PASCAL precludes the recognition of lines within the source input. To include object characters in character strings is feasible and would be a trivial change.

---

Requirement: H6

Key words will be reserved, be very few in number, be informative, and not be usable in contexts where an identifier can be used.

---- T ----

PASCAL contains only 30 reserved keywords which is considered very few for a language of this power. The predefined procedure identifiers are not included in this count.

---

Requirement: H7

The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, be introduced by a single (or possibly two) language defined characters, permit any combination of characters to appear, be able to appear anywhere reasonable in programs, automatically terminate at end of line if not otherwise terminated, and not prohibit automatic reformatting of programs.

---- P+ ----

Single comment convention . . . . . T

Distinguishable from code . . . . . P

Introduced by one or two characters . . . . . T

Contain any character . . . . . P+

Appear anywhere reasonable . . . . . T

Terminate of end-of-line . . . . . F

Not prohibit reformatting . . . . . T

Comments in the PASCAL language may appear any two lexical units (i.e., identifiers, numbers, operators, etc.), are introduced and

terminated by unique symbols, {and} respectively, thus precluding the use of the terminating symbol, }, as a comment character. As a result, it is probable that to a human reader comments are not "easily distinguishable from code". This subrequirement appears to conflict with that requiring "an combination of characters to appear". However compilers, obviously, would have no problem in distinguishing comments from source code.

Comments are not terminated by an end-of-line since the free format nature of the language does not recognize the end-of-line entity.

---

Requirement: H8

The language will not permit unmatched parenthesis of any kind.

---- T ----

---

Requirement: H9

There will be a uniform referent notation.

---- T ----

Entire variables are referenced by the variable identifier name. A component of a variable is referenced by the denotation of the variable identifier followed by a selector specifying the component. The form of the selector depends on the structuring type of the variable. However, for each structuring type references are consistent and this evaluator assumes that this is the intent of this requirement.

---

Requirement: H10

No language defined symbols appearing in the same context will have essentially different meanings.

---- T ----

The grammar of PASCAL is truly context free.

Requirement: I1

There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made irrevocable when the language is defined or explicitly in each program.

---- T ----

The structure of PASCAL requires all identifiers to be defined prior to reference and each executable statement is entirely explicit thus eliminating all of the traditional defaults. However, the collating sequence of the alfa character set is not a function of the language and will differ from one implementation to the other. This characteristic could be considered an unalterable default though it is unlikely to have anymore than a minor impact if the programming authorities are aware of the collating sequence for a specific implementation.

---

Requirement: I2

Default capabilities will be provided for special capabilities affecting only object representation on other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

---- U ----

This requirement appears to be primarily addressing the internal algorithms of a compiler which affect the object representation of data and object code generated but do not affect the functional logic. The PASCAL language definition does not address this subject.

---

Requirement: I3

The user will be able to associate compile-time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

---- P- ----

Although the language includes the feature of associating data

constants with compile-time identifiers, as defined in section D1 of this report, however, those identifiers may only be used locally and do not address the object machine environment. The required features could be added to the language at modest cost, however, the cost of a given system would be closely related to the characteristics of that system.

---

Requirement: I4

The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compiler time variables. In such cases, the conditional will be evaluated at compile-time and only the selected path will be compiled.

---- T/U ----

The PASCAL language contains the necessary features to satisfy this requirement (i.e., compile-time identifiers and case statements), however, it is not defined if compile-time evaluation of "constant" expressions and conditional compilation is a language feature. In general, this decision would be left to the compiler designer. It would, of course, be trivial to necessitate such features by including them in the language definition.

---

Requirement: I5

The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

---- T ----

This language, like its predecessor ALGO-60 contains a minimum base language from which more complex components may be constructed. Declarative statements are of two (2) basic types; data declarations and procedure/function declarations. Executable statements are of eight (8) basic types; assignment, procedure invocation, if, case, while, repeat, for, and go to resulting in a total of ten (10) basic statements. Though not a theoretical minimum, the power realized by this choice of statements would appear to be close to optimum.

The design objectives of PASCAL included; systematic structure, flexibility of program and datat structuring, and efficient implementability. This reviewer feels that those goals have been met.

---

Requirement: I6

Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language.

---- F ----

Information pertaining to translator dependent limits are not definable within the language. The effort to include a translator environment definition would, in most instances, be major.

Requirement: I7

Language restriction which are inherently dependent only on the object environment will not be built into the language definition or any translator.

---- T ----

As with other requirements concerning translator and object environments, information was not available from the language definitions. The language itself does not contain any restrictions which are dependent upon the object environment, however, it is not unreasonable to assume that translators exist with such restrictions.

Requirement: J1

The language and its translators will not impose run-time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

---- P ----

The source material used for this evaluation was limited to a language definition and did not address virtual or real translators. It is this reviewers opinion that the efficiency of the object representation is, to no small extent, a function of the ingenuity of the translator designer and the environment in which the object code will be invoked. However, the highly structured nature and syntactic consistency of the language will provide a hospitable environment for implementing efficient and reliable translators on presently available computers.

---

Requirement: J2

Any optimizations performed by a translator will not change the effect of the program.

---- T ----

As previously explained, information performing to specific translators was not available to this reviewer. It would, however, appear to be mandatory for any translator optimization that the program logic not be effected.

---

Requirement: J3

The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

---- P ----

PASCAL does not support machine language insertions, however, extending the standard procedure set would provide a mechanism for referencing encapsulated machine routines. Such routines would require a detailed knowledge of the parameter handling protocal and thus would be a function of the translator implementation and the run-time executive environment.

---

Requirement: J4

It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will distinct form the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

---- F ----

The physical description of the object representation of composite data structures cannot be specified from within the source of PASCAL programs. It is, however, feasible that a translator could include optimization based upon the logical description and usage.

The anticipated effort of including explicit physical specifications of data objects would be major.

---

Requirement: J5

The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and closed routine of the same description will have identical semantics.

---- F ----

The PASCAL language definition implies that all procedure definitions will be implemented as closed subroutines thus this requirement is not met. The inclusion of an open routine facility would be moderate if it were explicitly defined as such. However, if the translator were required to optimally determine an open or closed implementation the impact would be major.

# PASCAL EVALUATION SUMMARY

This section provides a top level summary of the evaluation of the PASCAL language. Those functions of the language which generally satisfy the HOL specification are highlighted, as are the deficiencies. Deficiencies are further classified into those which would require major and those which would require minor modification to satisfy the HOL requirements.

The development of PASCAL was based on two principal aims. The first was to make available a systematic, disciplined programming language based on a minimum number of fundamental concepts which were clearly and naturally reflected by the language. The second was to develop implementations of the language which were both reliable and efficient on presently available computers, thus dispelling the commonly accepted notion that useful languages must be either slow to compile or slow to execute, and the belief that any non-trivial system is bound to contain mistakes forever. A further airm was to provide data structures and functions to make it possible to solve both commercial and scientific problems to help erase the mystical belief that those disciplines must be segregated.

PASCAL includes the following characteristics which satisfy the requirements of the HOL specification:

Concise, well defined syntax providing maximum structural flexibility with the minimum of primitives and syntactic structures. Few keywords are required; word abbreviations are not permitted and language statements are free format.

Strongly typed identifiers, including pointer types, with well defined semantics which generally prohibit implicit data type conversions; extensible data types by enumeration, sub-ranging and powersets.

Structured complex composite data structures, i.e., arrays of records, etc., providing almost unlimited compositions of the primitive data types.

Block structured environment providing clearly discernible scopes of activity and limited recursive capabilities.

Minimum set of pre-defined standard functions and procedures which may readily be extended to optimize system performance for specific purposes.

Universal usage of expressions, of appropriate type, wherever a value is required.

Minimization of side effects by limiting identifier references in functions to local identifiers and disallowing assignments to formal parameters.

With only one exception, defaults are not permitted.

Language structure is such that efficient implementations should be readily obtained.

Conversely, PASCAL includes the following characteristics which either conflict or are deficient.

Absence of fixed-point data types.

Absence of range and resolution for floating point data elements.

Alternate record structures are selected by a single "tag" field in conflict with the HOL requirement for selection by a general Boolean expression.

Relaxation of type checking for assignments when dependent variable is real and expression is integer or integer subrange and where expression is subrange of dependent variable type.

Variable number of arguments to procedures is not supported.

Absence of explicit control structures for parallel processing, exception handling and asynchronous interrupts.

Language does not address real-time control functions, i.e., control path delay, etc.

Apparent absence of conditional copilation facilities based upon compile time variables specifying object computer environment.

Physical description of object data representation is not supported.

Open subroutines are not supported.

Recursive subroutines are not explicitly defined but are only implied by loca references.

Modification of PASCAL to more fully support the HOL specification would be feasible. As detailed in this report, some of the modifications would be minor, while others would be relatively major efforts, and costly. Those in the former group include; fixed point data types, range and resolution specification, compile or load time evaluation of constant expressions, etc. Those in the latter group include: extension of operators to new data types, seperation of allocation and access scopes, parallel processing and selection of open/closed subroutines. In addition to these latter items, there are a number of requirements which are in conflict with the basis of PASCAL and could not readily be solved without changing the concept of that language. The most significant of these are those concerning the treating of subrange types as of the same general type as the base type and the concept of generic procedures where the formal parameters are not fully specified. It is believed by this reviewer that the intent of the former is satisfied by relaxing type checking between base and subrange types. The latter poses more of a problem since it is difficult to envisage how formal parameter specifications could be made optional without being inconsistent with the type checking convention of the language and in fact, the general HOL requirements of type consistency.

In conclusion, it is the belief of this reviewer that PASCAL satisfies most of the salient point of the HOL requirement. With modifications and the acceptance of meeting the intent of some HOL requirements though not necessarily explicitly PASCAL should encompass the total HOL requirements.

# EVALUATION OF

# CS-4

Prepared by

RLG Associates, Inc.
11250 Roger Bacon Drive
Suite 16
Reston, Virginia 22090

December 3, 1976

This report presents an evaluation of the basic CS-4 language with the requirements listed in the "Tinman". (DOD requirements for high order computer programming languages, "Tinman" - 1 March, 1976, Section IV) For purposes of comparison CS-4 is considered to be defined by:

CS-4 Language Reference Manual and Operating System Interface.
Intermetrics,inc.,Cambridge,Mass.,October,1975.

There are 78 language requirements listed in section 4 of the "Tinman". This report compares basic CS-4 with each individual requirement. A summary of the degree to which the language satisifies each requirement is presented.

The introductory paragraph of each "Tinman" requirement is included as the leading section for each requirement evaluation.

Symbols placed beside each individual requirement indicate the degree to which the language meets the requirement. The symbols and their meaning are as follow:

T - Totally meets requirement

P - Partially meets requirement

F - Does not meet requirement

U - Unknown from available documentation

A summary of the evaluation is included as the last section of this report. Merits and failures of the language as it currently is defined are discussed. Also discussed are the potential language modifications that can be made to support DoD "Tinman" requirements.

Note: In the following text references to "CS-4" always imply basic CS-4. Any references to full CS-4 will be explicit.

## A. DATA AND TYPES

Requirement: A1

The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source program.

---T---

CS-4 totally meets this requirement.

---

Requirement: A2

The language will provide data types for integer, real (floating point and fixed point), boolean and character and will provide arrays (i.e., composite data structures with indexible components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

---P---

Integer.....T
Fixed point.....F
Floating point.....T
Character string.....T
Boolean.....T
Arrays.....T
Records.....T

CS-4 does not provide a seperate fixed-point mode. The CS-4 mode FRACTION provides for the special case of a floating real with an exponent of zero.

---

Requirement: A3

The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

---T---

CS-4 meets this requirement totally for all numeric data modes.

---

Requirement: A4

Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.

---F---

CS-4 does not support fixed point typed data.

---

Requirement: A5

Character sets will be treated as any other enumeration type.

---F---

CS-4 provides for a single character set (revised ASCII).

---

Requirement: A6

The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

---T---

Number of dimensions fixed at compile time.....T
Type fixed at compile time.....T
Lower subscript bound fixed at compile time....T
Subscripts from contiguous range.....T
Subscripts from enumeration type.....T
Upper subscript bound fixed at scope entry.....T

CS-4 totally meets this requirement.

---

Requirement: A7

The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

---T---

CS-4 totally meets this requirement.

---

## B. OPERATIONS

Requirement: B1

Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

---T---

Variable declaration for all data types.....T
Encapsulated type declaration.....T
Array or record declaration.....T

CS-4 totally meets this requirement.

---

Requirement: B2

The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

---T---

CS-4 totally meets this requirement.

---

Requirement: B3

Relational operations will be automatically defined for numeric data and all types defined by enumeration.

---T---

CS-4 totally meets this requirement.

---

Requirement: B4

The built-in arithmetic operations will include: addition, subtraction, multiplication, division with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

---P---

Addition.....T
Subtraction.....T
Multiplication.....T
Division (with real result).....T
Negation.....T
Exponentiation.....T
Division with remainder.....F

CS-4 provides the majority of the arithmetic operations specified in B4 with the following exceptions:

(a). The CS-4 operation IDIV provides for integer division of the various numeric types with no capability for access of remainder.

(b) Fixed point operations are not supported (see requirement A2).

---

Requirement: B5

Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

---T---

CS-4 totally meets this requirement.

---

Requirement: B6

The built-in boolean operators will include "and", "or", "not", and "nor". The operations "and" and "or" will be evaluated in short circuit mode.

---T---

Short circuit "and".....T
Short circuit "or".....T
Not.....T
Nor.....T

CS-4 totally meets this requirement.

---

Requirement: B7

The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

---T---

Assignment between records and arrays.....T
Scalar operations on arrays.....T

CS-4 totally meets this requirement.

---

Requirement: B8

There will be no implicit type conversions but no conver-

sion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

---P---

Explicit conversions.....U
No implicit conversions.....T
Explicit between scale factors.....F

Implicit conversion between modes in CS-4 is not premitted. No explicit conversion operations are defined in the available documentation.

---

Requirement: B9

Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

---T---

CS-4 totally meets this requirement.

---

Requirement: B10

The base language will provide operations allowing programs to interact with files, channels or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

---F---

"The CS-4 Operating System Interface" provides extensive I/O operations, these facilities however are extensions of the basic language definition and therefore are not included in this evaluation. No I/O operations are defined for basic CS-4.

---

Requirement: B11

The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

---T---

CS-4 totally meets this requirement.

---

## C. EXPRESSIONS AND PARAMETERS

Requirement: C1

Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

---T---

CS-4 totally meets this requirement.

---

Requirement: C2

Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

---P---

Unambiguous presentation.....T
Few precedence levels.....T
Require explicit parentheses.....F
No user defined levels.....T

Explicit parentheses are not required in expressions involving operators of equal precedence.
Full CS-4 provides facilities for defining new pre-,post-,and infix operators and attaching precedence to their evaluation. Existing operator precedence cannot be redefined.

---

Requirement: C3

Expressions of a given type will be permitted anywhere in source programs where both constants and references to

variables of that type are allowed.

---T---

CS-4 totally meets this requirement.

---

Requirement: C4

Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.

---T---

CS-4 totally meets this requirement.

---

Requirement: C5

There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types for exception handling, for parallel processes, for declaration, or for built-in operators. There will be no special operations (e.g., array structuring) applicable only to parameters.

---T---

Consistency in parameter rules.....T
No special parameter operations.....T

CS-4 totally meets this requirement.

---

Requirement: C6

Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

---T---

Dimensions determined at compile time.....T
Size and subscripts can be passed.....T
Type agreement for actual and formal parameters.....T

CS-4 totally meets this requirement.

---

Requirement: C7

There will be four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

---P---

Parameter constants.....T
Parameter variables.....T
Exception conditions.....F
Procedure parameters.....T

CS-4 does not provide a formal parameter class for exceptional condition control.

---

Requirement: C8

Specification of the type, range, precision, dimension, scale and format of parameters will be optional on the formal side. None of them will be alterable at run time.

---T---

Optional properties.....T
Fixed at run time.....T

CS-4 totally meets this requirement.

---

Requirement: C9

There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

---F---

Variable number of arguments.....F
All but constant number same type.....F
Number fixed at compile time.....F

CS-4 does not support procedures with a variable number of arguments. Full CS-4 does provide the necessary mechanisms for this requirement.

---

## D. VARIABLES, LITERALS AND CONSTANTS

Requirement: D1

The user will have the ability to associate constant values of any type with identifiers.

---T---

CS-4 totally meets this requirement.

---

Requirement: D2

The language will provide a syntax and a consistent interpretation for literals of built-in data types. Numeric literals will have the same value (within the specified precision) in both programs and data (input or output).

---T---

Literals of built-in data types.....T
Consistency in value.....T

CS-4 totally meets this requirement.

---

Requirement: D3

The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

---T---

Declare initial values.....T
Allocation scope initialization.....T
No default values.....T

CS-4 totally meets this requirement.

Requirement: D4

The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

---P---

Range specification is required for all numeric modes (specifictaion may be implicit).
Fixed point typed data is not supported (see note on re-

quirement A2).

---

Requirement: D5

The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.

---T---

CS-4 totally meets this requirement.

---

Requirement: D6

The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

---T---

Shared/recursive substructure capability.....T
Variable/record/array component handling.....T
Typed pointer characteristic.....T
Allocation never wider than access.....T

The CS-4 pointer mechism is limited to dynamicly allocated data (data allocated in HEAP storage via the *ALLOCATE* statement).

---

## E. DEFINITION FACILITIES

Requirement: E1

The user of the language will be able to define new data types and operations within his programs.

---T---

CS-4 totally meets this requirement.

---

Requirement: E2

The "use" of defined types will be indistinguishable from built-in types.

---T---

CS-4 totally meets this requirement.

---

Requirement: E3

Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

---T---

CS-4 totally meets this requirement.

---

Requirement: E4

The user will be able, within the source language, to extend existing operators to new data types.

---T---

CS-4 totally meets this requirement.

---

Requirement: E5

Type definitions in the source language will permit definition of both the class of data objects comprising the types and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

---T---

CS-4 totally meets this requirement.

---

Requirement: E6

The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of

disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

---T---

Enumeration.....T
Cartesian products.....T
Discriminated union.....T
Power set of enumeration type....T

CS-4 contains extensive data type definition facilities. Enumeration and power set definition can be applied through the use of the RANGE qualifier.

---

Requirement: E7

Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.

---T---

Does not permit free unions.....T
Does not permit subsetting.....T

CS-4 totally meets this requirement.

---

Requirement: E8

When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

---T---

CS-4 totally meets this requirement.

---

## F. SCOPES AND LIBRARIES

Requirement: F1

The language will allow the user to distinguish between scope of allocation and scope of access.

---T---

CS-4 totally meets this requirement.

---

Requirement: F2

The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

---T---

CS-4 totally meets this requirement.

---

Requirement: F3

The scope of identifiers will be wholly determined at compile time.

---T---

CS-4 totally meets this requirement.

---

Requirement: F4

A variety of application-oriented data and operations will be available in libraries and easily accesible in the language.

---U---

Variety of data and operations.....U
Easily accesible.....U

The compostion and extent of application oriented libraries is not defined for CS-4 at this time. The creation of such libraries should present no great problems at the time CS-4 is finally implemented.

---

Requirement: F5

Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

---U---

Accesible at compile time.....U
Other source language routines.....U

See note on requirement F4.

---

Requirement: F6

Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

---U---

Libraries and compools indistinguishable.....U
Hold anything definable in language.....U
Many levels of access.....U
Many specialized subsets.....U

See note on requirement F4.

---

Requirement: F7

The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

---I---

The MSTRUCTURE mode of CS-4 satisfies this requirement.

---

## G. CONTROL STRUCTURES

Requirement: G1

The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

---P---

Sequential control.....T
Conditional control.....T
Iteration.....T
Recursion.....F
Parallel processing.....F
Exception handling.....T
Asyncronous interrupts.....T

CS-4 provides all of the standard sturctured control mechanisms with the exception of parallel processing and recursion. Full CS-4 provides for the inclusion of control structures for both parallel processing and recursion.

---

Requirement: G2

The source language will provide a "go to" operation applicable to program labels within its most local scope of definition.

---T--

CS-4 totally meets this requirement.

---

Requirement: G3

The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

---T---

Based on boolean expression.....T
Based on type from discriminated union.....T
Based on computed choice.....T

CS-4 totally meets this requirement.

---

Requirement: G4

The iterative control structure will permit the termination condition to appear anywhere in the loop, will re-

quire control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g. fixed number of iteration or elements of an array exhausted).

---P---

Termination anywhere in loop.....T
Local control variables.....F
Entry at loop head only.....T
Efficient and clear simple cases.....T
Control value available at termination.....T

In CS-4 iterative control variables are not treated specially.

---

Requirement: G5

Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

---F---

Recursive and nonrecursive routines.....F
Explicitly specify recursive routines.....F
No nested recursive procedures.....F

CS-4 does not provide for recursion in any form. Full CS-4 supplies facilities for recursive routines which meet or exceed requirement G5.

---

Requirement: G6

The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

---F---

CS-4 has no parallel processing facilities. full CS-4,however, provides extensive parallel processing mechanisns which meet or exceed requirement G6.

---

Requirement: G7

The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

---T---

CS-4 totally meets this requirement.

---

Requirement: G8

There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

---F---

The necessary features to satisfy this requirement are contained in "the CS-4 Operating System Interface" or within full CS-4. Basic CS-4 does not include the required mechanisms.

---

## H. SYNTAX AND COMMENT CONVENTIONS

Requirement: H1

The source language will be free format with an explicit statement separator, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

---T---

Free format with statement separator.....T
Mnemonically significant identifiers.....T
Conventional forms.....T
Simple uniform grammar.....T
No special case notations.....T
No abbreviation of keywords or identifiers....T
Unambiguous grammar.....T

Although CS-4 syntax is quite extensive, it seems to be highly readable.

---

Requirement: H2

The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedence.

---T---

CS-4 totally meets this requirement.

---

Requirement: H3

The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

---P---

CS-4 uses the 94 printing characters of 128 character revised USACII character set (USA standard X3.4-1968). modification of the language to accept the 64 character subset should be trivial.

---

Requirement: H4

The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break

character for use internal to identifiers and literals.

---T---

Literals for numbers.....T
Character strings.....T
Break character.....T

CS-4 totally meets this requirement.

---

Requirement: H5

There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

---P---

CS-4 lexical units may not be continued across lines. at present there is no provision for inclusion of object characters in literal strings. The addition of this feature should be trivial.

---

Requirement: H6

Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

---T---

given the extent of the language, keywords are acceptably few in number and all are reserved.

---

Requirement: H7

The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

---P---

Single comment convention.....F
Distinguishable from code.....T
Introduced by one or two characters.....T
Contain any charcter.....P
Appear anywhere reasonable.....T
Terminate at end of line.....P
Not prohibit reformatting.....T

CS-4 contains two comment conventions:

(1) The precent character (%) may be used to indicate a commen terminates on end of line.

(2) The bracket notation ( { ,} ) form requires both comment delimiters to be present. termination is not automatic on end of line.

also each comment delimiter within the comment string must be matched.

---

Requirement:H8

The language will not permit unmatched parentheses of any kind.

---T---

CS-4 totally meets this requirement.

---

Requirement: H9

There will be a uniform referent notation.

---T---

CS-4 totally meets this requirement.

---

Requirement: H10

No language defined symbols appearing in the same context will have essentially different meanings.

---T---

CS-4 totally meets this requirement.

---

## I. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

Requirement: I1

There will be no defaults in programs which affect the program logic.That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

---T---

CS-4 totally meets this requirement.

---

Requirement: I2

Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

---T---

CS-4 totally meets this requirement.

---

Requirement: I3

The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

---P---

CS-4 does not include provisions for a special class of "compile time" variables. Instead any variable who's value is known at compile time (via initialization) may be utilized in compile time functions. Specification of the object machine configuration is accomplished through the use of %STRUCTURE specifications.

---

Requirement: I4

The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

---P---

CS-4 provides for compile time evaluation of expressions composed of known values. There are,however, no mechanisms whereby selective compilation may occur. Full CS-4 supplies the necessary compile time control structures for meeting this requirement.

---

Requirement: I5

The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

---T---

CS-4 is based upon the kernel language ELS-K.

---

Requirement: I6

Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

---T---

CS-4 totally meets this requirement.

---

Requirement: I7

Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

---T---

CS-4 totally meets this requirement.

---

## J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

Requirement: J1

The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

---U---

The ability or inability of a given CS-4 translator to produce efficient object code is an open question at this time. Certainly the CS-4 language designers desire efficiency of object code. Such a property,however,cannot be defined in the design of a language directed at a number of computers possesing a variety of architectures.

---

Requirement: J2

Any optimizations performed by the translator will not change the effect of the program.

---T---

CS-4 meets this requirement in the sense that it is a stated design goal of the language. In reality the determination of this question would be somewhat dependant upon the specific implementation of a given CS-4 translator.

---

Requirement: J3

The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

---T---

CS-4 totally meets this requirement.

---

Requirement: J4 It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapslated and will distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

---P---

Specify object presentation.....T
Specify time/space tradeoff.....U

Time/space tradeoff specification is not addressed in the available documentation.

---

Requirement: J5

The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and closed routine of the same description will have identical semantics.

---T---

CS-4 totally meets this requirement.

---

## CS-4 EVALUATION SUMMARY

This section presents a summary of the evaluation of basic CS-4 against the DOD "Tinman" requirements. The major strengths and weaknesses of the language in relation to the requirements will be detailed and appraisal given as to the overall desireability of utilizing CS-4 as a base language for future HOL developement.

The major strengths of CS-4 are:

**Strong data typing.** All data modes in CS-4 are rigidly typed as to data characteristics and access. In addition type checking is enforced during runtime as well as compile time. such a trait significantly increases the possiabilities of producing reliable programs.
**Extensive and logical control structures.** CS-4 provides a concise set of control mechanisms sufficient for structured program construction without being overly elaborate.
**Extensiablility.** CS-4 contains features for the definition of new data modes and for the utilization of these extended modes in a straightforward manner.
**Machine independance.** CS-4 is a highly machine independant language with a set of facilities for mating the language to a specific machine. Whether the facilities provided are sufficient for a tight fit is an open question that can only be resolved by implementation.
**Levels of usage.** CS-4 provides facilities whereby the language may be used on several levels. This multileveling includes the enabling/disabling of GOTO usage,run-time range checking and other language features. This type of usage scheme would seem well suited to the stepped developement of both programs and programmer skill.

The major weaknesses of CS-4 are:

**Lack of a fixed-point data mode.** The CS-4 designers rationalized the exclusion of fixed-point as being both unnecessary due to the drop in cost of floating point hardware and inefficient because of the necessity of shifting for decimal point alinement in arithmetic operations. The validity of these arguments would seem to hold for medium to large scale systems while becoming suspect when applied to most mini systems.
**Lack of I/O facilities in the base language.** Basic CS-4 contains no I/O facility definitions. Instead I/O mechanisms are defined within the context of "The CS-4 Operating System Interface", a set of procedure calls which define,in addition to I/O services,parallel processing constructs and other miscelaneous operating system functions. Embedded within this "interface" is the definition of a complete file system structure. The inclusion of such a specification as part of the basic language would be unacceptable as being too extensive ( the language designers have in effect defined an operating system for their language).

CS-4 is an unimplemented language. This might not be fairly considered a defect in relation to "Tinman" requirements,however in relation to the overall consistancy of the language it would seen that much benifit could be gained by the feedback resulting from a trail implementation.

No recursive procedures. CS-4 procedures have no capability for recursion. This defect has been corrected in full CS-4.

No parallel processing capability. Parallel processing constructs ,like the I/O mechanisms of CS-4, are embedded within the "Operating System Interface".

Lack of concise conversion procedures. CS-4 documentation is quite specific that no implicit conversion between unlike data modes will occur. No explicit conversion mechanisms are specified however.

In summary CS-4 is an extensive,near state-of-the-art language which contains a great number of the traits outlined in the "Tinman" requirements. The extended version of the language,full CS-4,contains nearly all of the necessary features for meeting the requirements. On the negative side CS-4 has several serious defects,all of which seem to be outgrowths of the fluid nature of the language. If this lack of consistancy can be corrected,either through feedback from trail implementations or a more unified language design effort, CS-4 would make a fine choice for a HOL baseline.

# EVALUATION OF

# TACPOL

Prepared by

RLG Associates, Inc.
11250 Roger Bacon Drive
Suite 16
Reston, Virginia 22090

December 3, 1976

This report presents an evaluation of the TACPOL language with the requirements listed in the "TINMAN". (DOD Requirements for High Order Computer Programming Languages, "TINMAN" - 1 March 1976, Section IV). For purposes of comparison TACPOL is considered to be defined by:

> CPCEI SPECIFICATION FOR COMPILER/ASSEMBLER FOR FIRE DIRECTION SYSTEM, Spec. No. EL-CG-00043082C, Doc. No. 595909-600C, Volume 1 of 2, Appendix 10 (FORMAL DEFINITION OF TACPOL)

There are 78 language requirements listed in section 4 of the "TINMAN". This report compares TACPOL with each individual requirement. A summary of the degree to which the language satisfies each requirement is presented.

The paragraph number of each "TINMAN" requirement is included as the leading section for each requirement evaluation.

Symbols placed beside each individual requirement indicate the degree to which the language meets the requirement. The symbols and their meaning are as follows:

T - Totally meets requirement

P - Partially meets requirement

F - Does not meet requirement

U - Unknown from available documentation

A summary of the evaluation is included as the last section of this report. Merits and failures of the language as it currently is defined are discussed. Also discussed are the potential language modifications that can be made to support DOD "TINMAN" requirements.

## A. DATA AND TYPES

Requirement: A1

The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile-time and unalterable at run-time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source program.

--- T ---

TACPOL is a strongly typed language. All quantities (units of storage which hold a value) must be one of four possible types: short numeric; long numeric; character string; bit string.

Quantity types cannot be altered at run-time but value conversions can be made by means of several "intrinsic procedures" (explicit conversion functions which may be used in the body of a TACPOL program and which have meaning independent of the definitions given by the programmer). These procedures (listed below) make appropriate conversions on the value of the argument of the procedure.

| | |
|---|---|
| SHORT (X) | long numeric to short numeric, character string to short numeric, or bit string to short numeric (specific conversion selected depends on compile-time of X) |
| LONG (X) | short numeric to long numeric, character string to long numeric, or bit string to long numeric |
| CHAR (X) | short numeric to character string, long numeric to character string, bit string to character string |
| BIT (X) | short numeric to bit string, long numeric to bit string, or character string to bit string |

TACPOL does support a cell structure which is designed to permit an equivalence of storage usage (i.e., overlay) at run-time. This is used principally to buffer I/O records of different formats. Depending on the run-time checking available (undefined in TACPOL), it might be possible to subvert type checking.

---

Requirement: A2

The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexible components of homogeneous type) and records (i.e., composite data structures with labeled components of hetero-

geneous type) as type generators.

--- P ---

Integer . . . . . P
Fixed Point . . . . . P
Floating Point . . . . . T
Character string . . . . T
Boolean . . . . . T
Arrays . . . . . T
Records . . . . . P

TACPOL supports the four data types described in A1. In the case of numerics, TACPOL includes the ability to state precision (number of bits representing the most significant part of the value) and scale (where to place the binary point). Short numeric permits a precision of from 1 to 31 bits; long numeric from 32 to 62 bits. The scale must be between -127 and +127. Operations on numerics take into account the precision and scale of each operand.

Although all numbers are stored internally in one word (32 bits) or two word (64 bit) quantities with the appropriate scale factors, the programmer may code number values as integers, fixed point, or floating point and they will be treated properly both on input and on output.

TACPOL also provides arrays (up to 3 dimensions) as quantity structures which may be declared and designated.

The concept of record is well understood in TACPOL, although a strong distinction is made between an externally recorded record (without much detail) and the well defined or typed quantities in main memory from/to which records are written/read (normally cells).

---

Requirement: A3

The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

--- P ---

As discussed in A2, TACPOL does support precision specification on individual quantities but not any sort of global scoping of precision requirements.

All operations must take into account the individual precision specifications of each operand.

In terms of hardware independence, TACPOL does not specify a particular machine, but it seems strongly oriented toward machines with 8-bit bytes, 32-bit words, whose internal character set is 8-level ASCII.

Global scoping of precision could be added to TACPOL without much difficulty, through the addition of new syntax.

---

Requirement: A4

Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile-time. Scale factor management will be done by the compiler.

--- T ---

Within the limits of available numeric precision, TACPOL fully supports the scaled integer feature.

---

Requirement: A5

Character sets will be treated as any other enumeration type.

--- F ---

TACPOL does not support the definition and ordering of characters by the source program. Characters are said to be in ASCII, one 8-bit byte equals ASCII character.

---

Requirement: A6

The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile-time. The upper subscript bound will be determinable at entry to the array allocation scope.

--- P ---

Number of dimensions fixed at compile-time . . . . T
Type fixed at compile-time . . . . . T
Lower subscript bound fixed at compile-time . . . T
Subscripts from contiguous range . . . . . T
Subscripts from enumeration type . . . . . T
Upper subscript bound fixed at scope entry . . . . F

TACPOL supports one, two, and three dimensional arrays. The dimension of each array and the range of each subscript is fixed at compile (declaration) time. Subscript ranges must fall between one (always the lower bound) and N, where N is a number explicitly given at declaration time. N may be any type of numeric quantity which yields a value not less than one. A real quantity is truncated to the largest whole integer not greater than its value.

At designation, subscripts may be an expression which yields

a short numeric value within the declared range for that subscript; real values are reduced to integers as above.

Variable upper bounds on arrays are not supported.

---

Requirement: A7

The language will permit records to have alternative structures, each of which is fixed at compile-time. The name and type of each record component will be specified by the user at compile-time.

--- p ---

TACPOL places no restrictions on the use of alternative structures for records. Each file operation in TACPOL has associated with it a fully defined buffer data structure, with each element (scalar, array, table, group, cell) in the structure well defined and typed. There is nothing to prevent the programmer from reading one record into one structure and the next into another.

In addition, the programmer can declare a "cell" (a collection of sets of quantities) all of which are allocated storage in common (i.e., overlayed). Each set of quantities defines a potentially different data structure for a record. Even in this case, full type checking is done as the subcell quantities are used.

## B. OPERATIONS

Requirement: B1

Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

--- P ---

Variable declaration for all data types . . . T
Encapsulated type declaration . . . . . T
Array or record declaration . . . . . T
Array assignment . . . . . P

TACPOL supports referencing of quantities and assignment of values to any type of quantity or structure which can be declared. In the case of assignment, multiple quantities can be assigned a single value in one assignment operation, but multiple quantities (e.g., arrays) cannot be assigned multiple values in a single operation. The language fully supports the ability to reference any low level element in a complex data structure. Reference always retrieves the last assigned value.

---

Requirement: B2

The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

--- T ---

TACPOL supports the = relational operator which can be used to compare any two objects of the same type for identity. All four TACPOL-recognized types are supported.

Numeric values are compared algebraically, character string values character-by-character, left to right, and bit string values bit-by-bit, left to right.

---

Requirement: B3

Relational operations will be automatically defined for numeric data and all types defined by enumeration.

--- P ---

TACPOL supports all six relational operators on all four types which it recognizes.

TACPOL does not recognize unordered sets.

---

Requirement: B4

The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

--- P ---

Addition . . . . . T
Subtraction . . . . . T
Multiplication . . . . . T
Division (with real result) . . T
Negation . . . . . T
Exponentiation . . . . . T
Division with remainder . . . . P

TACPOL supports all of these operations.

All except for division with remainder are available in single syntactical operators. In order to do division with remainder, three operators are needed:

| | |
|---|---|
| normal real division | (A/B) |
| intrinsic procedure | TRUNC (C) |
| intrinsic procedure | REM (D) |

The first two can be confined to get the whole dividend of the division as follows:

DIV = TRUNC (A/B)

The remainder is obtained as follows:

REMAINDER = REM (A,B)

---

Requirement: B5

Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

--- P ---

Rounding takes place in TACPOL only when the intrinsic procedure, ROUND, is explicitly called by the programmer. Round operates only on the least significant part of the numeric value.

Truncation can be explicitly done in TACPOL by means of the intrinsic procedure TRUNC, which operates only on the least significant part of a numeric value.

Implicit truncation takes place during an assignment operation when the defined precision of quantity being assigned a value is less than that of the value derived from the assignment expression. Once again, only the least significant part of the numeric value is affected.

Rounding of floating point numbers takes place only when specifically called for. The implicit default is truncation.

---

Requirement: B6

The built-in Boolean operators will include "AND", "OR", "NOT", and "NOR". The operations "AND" and "OR" will be evaluated in short circuit mode.

--- P ---

Short circuit "and" . . . . . U
Short circuit "or" . . . . . U
Not . . . . . T
Nor . . . . . P

Short-circuit mode is not addressed in the TACPOL FORMAL DEFINITION.

Normal AND, OR, and NOT are supported by TACPOL.

NOR does not exist as an *explicit syntactical* element, although the identical effect of NOR can be achieved by means of the intrinsic procedure BOOL as follows (where X and Y are the bit variables being NORed, and the bit literal is the truth table):

BOOL (X,Y, '1000'B)

---

Requirement: B7

The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

--- P ---

Assignment between records and arrays . . . . P
Scalar operations on arrays . . . . . F

TACPOL supports the assignment of values of one set of quantities to a second set of quantities by means of the intrinsic procedure, MOVE (X,Y). There is no requirement that the two structures (e.g., arrays) be conformable; the length of the shorter of the two determines the scope of the operation.

The FORMAL DEFINITION does not define the scope or nature of

type checking in the MOVE procedure.

Scalar operations on arrays are not supported.

---

Requirement: B8

There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

--- P ---

Explicit conversions . . . . . P
No implicit conversions . . . . . T
Explicit between scale factors . . T

TACPOL supports a full range of type conversions (see answer to Requirement A1), for the types it recognizes.

Numeric values are all stored internally as one or two word binary quantities with a scale factor, hence numeric operations (within short or within long) always yield numeric values. The requirement for explicit conversion operations between integer, fixed point, and floating point internal data is undefined.

No implicit conversions are supported.

Numeric values can be rescaled (with no change in precision) by means of the intrinsic procedure, SCALE.

---

Requirement: B9

Explicit conversion operations will not be required between numerical ranges. There will be a run-time exception condition when any integer or fixed point value is truncated.

--- F ---

TACPOL does not support this function.

---

Requirement: B10

The base language will provide operations allowing programs to interact with files, channels or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be in-

stallation dependent.

--- P ---

TACPOL provides support for a variety of file declarations and file processing operations.

In TACPOL, files are named and declared to be either serial or direct (by key), partitioned or nonpartitioned, with concurrent or nonconcurrent I/O; a file is opened in one of three modes -- input, output, update.

Defined file operations include: OPEN, CLOSE, READ, WRITE, WRITE ENDFILE, REWRITE, DELETE, SPACE, REWIND, UNWIND. Exception condition processing on end-of-file, invalid record key, and invalid partition key is supported.

TACPOL does not consider devices or channels explicitly, only files. Assignment and reassignment of I/O devices is therefore undefined.

---

Requirement: B11

The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

--- F ---

Power sets are not defined in TACPOL.

## C. EXPRESSIONS AND PARAMETERS

Requirement: C1

Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

--- P ---

Numeric expression evaluation within TACPOL is done according to the following precedence order of operators (from highest to lowest precedence):

infix operators
exponentiation
multiplication/division
addition/subtraction

Boolean expression evaluation is done according to the following precedence:

relational operators
NOT
AND
OR
concatenation

Within any precedence level and for all other expressions, evaluation is performed left to right.

---

Requirement: C2

Which parts of an expression constitute the operands to each operation, within that expression, should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

--- P ---

Unambiguous presentation . . . . . P
Few precedence levels . . . . . T
Require explicit parentheses . . . . P
No user defined levels . . . . . T

TACPOL expression presentation is relatively straightforward, with the usual order of operator precedence. (See answer to Requirement C1.)

Parenthesis are permitted to reduce ambiguity or to alter precedence, but are not required except in the case of a relational subexpression within a Boolean expression, where the entire subexpression must be enclosed in parentheses.

Two unusual features of expression notation which (in this reader's opinion) detract from overall readability of expression are the operators:

(**) and (*)

which are forms of exponentiation and multiplication, but which create long numeric result values even though the operands are of the short numeric type.

---

Requirement: C3

Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

--- P ---

TACPOL meets this requirement in most cases, but there are some minor exceptions. For example, in the designation of a substring, the character count argument can only be an integer literal, not an expression.

---

Requirement: C4

Constant expressions will be allowed in programs where constants are allowed, and constant expressions will be evaluated before run-time.

--- P ---

In TACPOL, constant expressions can replace constants anywhere in the program, except in type specifications, array declarations, table declarations, value declarations, and in certain arguments of intrinsic procedures. These exceptions are not regarded as major shortcomings.

The evaluation of constant expressions before run-time is generally undefined in the FORMAL DEFINITION.

---

Requirement: C5

There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types for exception handling, for parallel processes, for declaration, or for built-in operators. There will be no special operations (e.g., array structuring) applicable only to parameters.

--- P ---

Consistency in parameter rules . . . . . P
No special parameter operations . . . . P

Generally, the usage of parameters appears similar. This is especially true in simple cases. However, as the user takes advantage of the more sophisticated data structures in TACPOL (e.g., cells), the rules for parameter usage start to

vary, depending on what is needed to describe the structure.

Parallel processing is not defined. Exception handling is defined only for file I/O operations and has a syntax all its own.

---

Requirement: C6

Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile-time. The size and subscript range for array parameters need not be determinable at compile-time, but can be passed as part of the parameter.

--- P ---

Dimensions determined at compile-time . . . . . T
Size and subscripts can be passed . . . . . F
Type agreement for actual and formal parameters . . . . T

In TACPOL, the dimensions of an array are fully determined at compile-time. This includes the number, size, and subscript range.

Size and subscript range cannot be passed as part of the parameter.

Formal and actual arguments must always agree in type.

---

Requirement: C7

There will be four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

--- P ---

Parameter constants . . . . . T
Parameter variables . . . . . T
Exception conditions . . . . . P
Procedure parameters . . . . . T

TACPOL does not classify formal parameters in a universal sense although it does meet some of the requirements stated.

When formal parameters are listed in a procedure declaration, they must also be defined by an imbedded parameter declaration which classifies each as a quantity (variable), value (constant), procedure parameter, or point (label).

Exceptions, per se, are defined only for file I/O, and permit the execution of a single statement in the event that

one of the three conditions -- end-of-file, invalid record key, invalid partition key -- are met. The structure for this is:

ON condition THEN statement ELSE statement

Other run-time conditions which can be checked are divide by zero, fixed overflow, or reference to a specified identifier name. The desire to check for such conditions is declared by the programmer. Upon occurrence, a "snap procedure" (mentioned but undefined in TACPOL) is to be implicitly invoked. The structure of an interface to such a procedure is not discussed in the FORMAL DEFINITION.

---

Requirement: C8

Specification of the type, range, precision, dimension, scale and format of parameters will be optional on the formal side. None of them will be alterable at run-time.

--- P ---

Optional properties . . . . . F
Fixed at run-time . . . . . T

TACPOL requires that all formal parameters in a declared procedure be themselves declared by type, precision, dimension, and scale. Range is implicit in short and long scalar definition. Format is not addressed.

Procedure parameters are not alterable at run-time.

---

Requirement: C9

There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile-time.

--- P ---

Variable number of arguments . . . . . F
All but constant number same type . . F
Number fixed at compile-time . . . . . T

TACPOL does not support variable numbers of arguments.

The number of required arguments for a procedure is fixed and fully determinable at compile-time.

## D. VARIABLES, LITERALS AND CONSTANTS

Requirement: D1

The user will have the ability to associate constant values of any type with identifiers.

--- T ---

TACPOL provides this capability.

---

Requirement: D2

The language will provide a syntax and a consistent interpretation for literals of built-in data types. Numeric literals will have the same value (within the specified precision) in both programs and data (input or output).

--- T ---

Literals of built-in data types . . . T
Consistency in value . . . . . T

TACPOL supports this capability.

---

Requirement: D3

The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

--- T ---

Declare initial values . . . . . T
Allocation scope initialization . . . . T
No default values . . . . . T

Using the value declaration statements, the user can declare the initial value of individual variables. If he does not, no default values are assigned.

---

Requirement: D4

The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

--- P ---

TACPOL requires that all numeric type variables be declared with a range which is composed of a precision (mandatory) and a scaling (optional, default is zero). The interpretation of range is for both compile-time and object time values (no difference is discussed).

TACPOL does not specifically address fixed point variables or their step size.

Within range specification, a precision of 1 to 31 bits implies short numeric type, and one of 32 to 62 bits implies long numeric type.

---

Requirement: D5

The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.

--- P ---

TACPOL supports this feature, except that user-defined types and ennumeration types are not defined.

---

Requirement: D6

The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

--- F ---

Shared/recursive substructure capability . . . . F
Variable/record/array component handling . . . . F
Typed pointer characteristic . . . . . F
Allocation never wider than access . . . . . F

The concept of a pointer mechanism (i.e., indirect addressing) is not defined in the FORMAL DEFINITION, except in one location (section 10.6.8, Proper Procedure Designations) where the metalinguistic element <ADDR DESIG> (one argument designation) is defined to be /<BIT EXPR>/. Nothing else is mentioned anywhere else. The implication is that a hard address can be physically built using Boolean operations. There appears to be no other mechanism in the language to deal with pointers.

## E. DEFINITION FACILITIES

Requirement: E1

The user of the language will be able to define new data types and operations within his programs.

--- F ---

TACPOL does not support these features.

---

Requirement: E2

The "use" of defined types will be indistinguishable from built-in types.

--- F ---

Defined types are not supported in TACPOL.

---

Requirement: E3

Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

--- P ---

TACPOL requires that many program components be defined explicitly. There are certain exceptions, however, such as:

- Scale factor defaults to zero on numeric quantities.
- Precision defaults to 31 on short numeric quantities.
- Data alignment defaults to PACKED or ALIGNED, depending on declaration type.
- For literals, E00 is default, and scaling defaults to 3.322 times the number of fractional digits in the literal.
- In substring operations, the optional count defaults to one.
- Several file-processing syntax elements can default to implicit values.
- All intrinsic procedures have default values and implicit range restrictions.

In addition, there are numerous implicit limits enforced by TACPOL, such as:

- The precision of numeric values.
- The maximum length of character strings (512) and bit strings (32).
- The maximum value of the step variable in a DO statement (32,767).

---

Requirement: E4

The user will be able, within the source language, to extend existing operators to new data types.

--- F ---

TACPOL does not support the definition of new types, and hence not the extension of existing operators of these types.

---

Requirement: E5

Type definitions in the source language will permit definition of both the class of data objects comprising the types and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

--- F ---

TACPOL does not support any of these features.

---

Requirement: E6

The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile-time.

--- F ---

TACPOL does not support the capability of having user-defined types. Only built-in types are allowed.

---

Requirement: E7

Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.

--- T ---

Does not permit free unions . . . . T
Does not permit subsetting . . . . . T

TACPOL meets this requirement since no method of type definition is permitted.

---

Requirement: E8

When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

--- F ---

TACPOL does not support user-defined types.

## F. SCOPES AND LIBRARIES

Requirement: F1

The language will allow the user to distinguish between scope of allocation and scope of access.

--- P ---

TACPOL explicitly addresses scope of allocation by means of a comprehensive set of data structures which can be declared.

Scope of access is not addressed explicitly, although it can be derived from data structure designations in the program. Explicit block structures with local scoping of variables are supported.

The cell structure in TACPOL permits several different substructures to be allocated the same physical storage.

---

Requirement: F2

The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

--- P ---

TACPOL does not support user-defined data types, nor does it distinguish between normal and "special" structures.

TACPOL does support block structures, with local scoping of variables.

---

Requirement: F3

The scope of identifiers will be wholly determined at compile-time.

--- T ---

TACPOL meets this requirement.

---

Requirement: F4

A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

--- P ---

Variety of data and operations . . . . P
Easily accessible . . . . . P

TACPOL supports the program use of COMPOOL data (although the FORMAL DEFINITION does not clearly state how). In addition, a variety of intrinsic procedures are available, although this set is predefined and not alterable by the user.

By means of the CALL statement, separately created external procedures can be accessed. Potentially, this feature could be used to access user libraries. This feature is not described in any detail in the FORMAL DEFINITION.

---

Requirement: F5

Program components not defined within the current program and not in the base language will be maintained in compile-time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

--- P ---

Accessible at compile-time . . . . . T
Other source language routines . . . P

Although the FORMAL DEFINITION does not describe exactly how, TACPOL can support separate program components which are CALLed as needed. If these components have been compiled by a TACPOL compiler using the CODE language feature, then they may be written in other source languages.

TACPOL does not, however, allow direct calling of external routines written in another source language without the CODE feature.

---

Requirement: F6

Libraries and compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

--- U ---

Libraries and COMPOOLS indistinguishable . . . U
Hold anything definable in language . . . . . U
Many levels of access . . . . . U
Many specialized subsets . . . . . U

Libraries are not explicitly addressed in the FORMAL DEFINI-

TION, although the ability to call an external procedure implies their existence. COMPOOLS are said to exist, but no further definition of COMPOOL or of language mechanisms to address them is given. Hence, the required features are undefined.

---

Requirement: F7

The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

--- P ---

TACPOL provides machine independent interface with peripheral equipment by means of its file processing features (READ/WRITE level). The "device" is designated only through some undefined external correspondence with the program file names. Some extension of this capability could provide an interface with other special hardware.

## G. CONTROL STRUCTURES

Requirement: G1

The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

--- P ---

Sequential control . . . . . T
Conditional control . . . . T
Iteration . . . . . T
Recursion . . . . . F
Parallel processing . . . . P
Exception handling . . . . . P
Asynchronous interrupts . . U

In addition to normal sequential execution, TACPOL supports the following control elements:

DO (WHILE) - iterative
GOTO - alter execution sequence
IF . . . THEN . . . ELSE - conditional

Recursion is undefined in TACPOL, and appears to be impossible since procedure bodies are physically inserted in-line (macro fashion) during compilation.

Parallel processing is not defined except for the ability to do concurrent I/O.

Exception handling is defined only for three cases of I/O processing - end-of-file, bad record key, bad partition key. In addition, condition checking can be performed on divide-by-zero, fixed overflow, and variable usage.

Interrupts are not defined explicitly in TACPOL. The WAIT operation causes processing to be suspended until all I/O operations on a file have ceased.

---

Requirement: G2

The source language will provide a "GOTO" operation applicable to program labels within its most local scope of definition.

--- T ---

TACPOL meets this requirement.

---

Requirement: G3

The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

--- P ---

Fully partitioned . . . . . F
Based on Boolean expression . . . . . T
Based on type from discriminated union . . F
Based on computed choice . . . . . T

The ELSE clause in the IF statement is optional.

The conditional expression in an IF statement must be a Boolean expression. The selection argument in the SWITCH intrinsic procedure (analogous to a CASE statement) must be a short numeric argument.

Discriminated unions are not defined in TACPOL.

---

Requirement: G4

The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity, or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

--- P ---

Termination anywhere in loop . . . . . F
Local control variables . . . . . U
Entry at loop head only . . . . . T
Efficient and clear simple cases . . . . . T
Control value available at termination . . U

Execution termination of a DO, either as a result of control variable value or of WHILE state, takes place at the top of the loop, since these conditions are only evaluated at that time.

Control variable must be a short numeric value, but is otherwise undefined in the FORMAL DEFINITION as being local or global. Whether the end value of the control variable is available is not defined in the FORMAL DEFINITION.

---

Requirement: G5

Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

--- P ---

Recursive and nonrecursive routines . . . . . P
Explicitly specify recursive routines . . . . . F
No nested recursive procedures . . . . . T

Recursive procedures are not defined in TACPOL, and appear to be impossible since procedure bodies are physically inserted in-line (macro fashion) during compilation.

---

Requirement: G6

The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

--- F ---

TACPOL does not support parallel processing.

---

Requirement: G7

The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

--- P ---

TACPOL supports exception handling only for three specific I/O cases -- end-of-file, bad record key, bad partition key. In addition, the following program conditions can be declared and (presumably) checked -- divide by zero, fixed overflow, variable usage. Exactly how these later conditions are processed, once detected, is not defined.

---

Requirement: G8

There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, and which permit asynchronous hardware interrupts to be treated as any other exception situation.

--- F ---

These features are not supported by TACPOL.

## H. SYNTAX AND COMMENT CONVENTIONS

Requirement: H1

The source language will be free format with an explicit statement separator, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple, uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

--- P ---

Free format with statement separator . . . . . P
Mnemonically significant identifiers . . . . . T
Conventional forms . . . . . P
Simple uniform grammar . . . . . P
No special case notations . . . . . P
No abbreviation of keywords or identifiers . . . T
Unambiguous grammar . . . . . T

TACPOL is basically of free format with an explicit separator (semicolon).

There are a few exceptions to this, however, including:

- The last statement in a CODE block must have ENDspace in columns 10-13
- Within certain control structures (e.g., DO, IF) the semicolon is not present in all cases.

It allows mnemonically significant identifiers, and utilizes generally conventional forms, except for some designations of complex data structures. At the top level, the grammar is uniform, but data structure designations can be far from simple or uniform.

Abbreviations of keywords are not permitted. The grammar seems to be unambiguous.

---

Requirement: H2

The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms, or define new infix operator precedence.

--- T ---

TACPOL meets these requirements.

---

Requirement: H3

The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

--- T ---

TACPOL uses a 50 character subset or the 64 character ASCII set.

---

Requirement: H4

The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

--- T ---

Literals for numbers . . . . . T
Character strings . . . . . T
Break character . . . . . T

TACPOL meets these requirements. The break character is the underscore.

---

Requirement: H5

There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

--- U ---

Neither of these requirements is defined in the TACPOL FORMAL DEFINITION.

---

Requirement: H6

Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

--- T ---

TACPOL recognizes 97 key words, all of which are fairly informative in context.

---

Requirement: H7

The source language will have a single uniform comment convention. Comments will be easily distinguishable from code,

will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

--- P ---

Single comment convention . . . . . T
Distinguishable from code . . . . . T
Introduced by one or two characters . . . . T
Contain any character . . . . . P
Appear anywhere reasonable . . . . . P
Terminate at end of line . . . . . F
Not prohibit reformatting . . . . . T

TACPOL comments begin with /* and end with */. They may contain any TACPOL character except / and *.

In TACPOL, comments may appear anywhere an arbitrary space may appear. Generally this is reasonable, although overzealous use of comments in certain contexts (e.g., between the keyword IF and the conditional expression which follows it) may make the program less, rather than more, readable.

There is no requirement in TACPOL that comments terminate at the end of a line. Only */ can terminate a comment.

---

Requirement: H8

The language will not permit unmatched parentheses of any kind.

--- T ---

TACPOL meets this requirement.

---

Requirement: H9

There will be a uniform referent notation.

--- P ---

In general, TACPOL uses the same notation for both function calls and identifier designations, i.e, the name of the function or the identifier followed by arguments in parentheses. However, the syntax of what appears in parentheses is highly dependent on the item (function or variable) being described.

---

Requirement: H10

No language defined symbols appearing in the same context

will have essentially different meanings.

--- P ---

Generally, TACPOL meets this requirement except for one case: the symbol = is used for both assignment and equality.

## I. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

Requirement: I1

There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

--- P ---

Generally, TACPOL meets this requirement. Certain defaults are allowed however. See answer to Requirement E3 for some examples.

---

Requirement: I2

Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

--- P ---

Generally, TACPOL meets this requirement. See I1 and E3 for exceptions.

---

Requirement: I3

The user will be able to associate compile-time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

--- F ---

TACPOL does not support these features.

---

Requirement: I4

The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile-time variables. In such cases the condition will be evaluated at compile-time and only the selected path will be compiled.

--- F ---

Although TACPOL does have a SWITCH capability, it is not used in the manner described.

---

Requirement: I5

The source language will contain a simple, clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

--- F ---

TACPOL contains no such base or kernel.

---

Requirement: I6

Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

--- P ---

The FORMAL DEFINITION of TACPOL explicitly denotes most of these restrictions, including the number of array dimensions and length of identifiers. Other translator restrictions (e.g., depth of nested parenthesis levels in expressions and maximum number of identifiers in a program) are not given in the FORMAL DEFINITION and cannot be explicitly defined b the user.

---

Requirement: I7

Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

--- T ---

TACPOL meets these requirements.

## J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

Requirement: J1

The language and its translators will not impose run-time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

--- U ---

Run-time considerations (including object code efficiency) are not defined in the TACPOL FORMAL DEFINITION.

---

Requirement: J2

Any optimizations performed by the translator will not change the effect of the program.

--- U ---

These considerations are undefined in the FORMAL DEFINITION.

---

Requirement: J3

The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

--- p ---

TACPOL meets most of this requirement with its CODE statement (which precedes a block of non-TACPOL statements).

TACPOL does not, however, support compile-time conditional statements of the sort described in Requirement I4.

---

Requirement: J4

It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

--- p ---

Specify object presentation . . . . F
Specify time/space tradeoff . . . . T

TACPOL meets the requirement of time/space tradeoff with its ALIGN/PACKED option in declarations.

TACPOL does not support user defined object presentation of composite data structures.

---

Requirement: J5

The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and closed routine of the same description will have identical semantics.

--- p ---

TACPOL expands all procedures in-line, and does not explicitly support closed procedures. There is, however, a suggestion that pre-compiled external procedures can be called; this feature is not explicitly stated or described, however.

## TACPOL Evaluation Summary

This section presents a summary of the findings of the evaluation of the TACPOL language. Strengths and weaknesses of the language are presented. Requirements for language modification necessary to meet DOD needs and functions are addressed.

TACPOL has a set of strengths which conform closely to the requirements of "TINMAN". Included in these are the following:

Well defined syntax -- The syntax of the TACPOL language cannot be altered by the user. The language is based on a relatively small set of fairly meaningful keywords, and these may not be abbreviated. The language is almost completely free format.

Strongly typed -- All identifiers used in a TACPOL program must be declared by type before they can be used. Type checking between parameters and arguments in procedures is enforced.

Block structured -- TACPOL supports user defined blocks, within which local scoping of identifiers is permitted.

Declaration and designation of variables, literals, and constants -- TACPOL meets virtually all of these requirements.

TACPOL also suffers from numerous significant weaknesses, primarily omission of certain required "TINMAN" features. These include:

Pointers are not supported.

User cannot define new types of operators or ennumeration sets.

There is no facility for describing the object environment or providing an external interface to the run-time machine.

Shortcomings exist in the definition of iterative and conditional control structures, including condition evaluation and a non-mandatory ELSE.

Recursion is not well defined.

Parallel processing is not supported, nor do any time dependent features exist.

Exception handling is inadequate.

The symbol = is used as an assignment as well as a relational operator.

There is a considerable amount of defaulting permitted,

not all of which is consistently applied.

The only TACPOL features which appear to exceed "TINMAN" specifications are:

The intrinsic procedure BOOL permits evaluation of two bit strings according to a "truth table".

The numbers and sophistication possible for data structures are considerable.

The syntax of TACPOL does appear to conform to the rules specified for the evaluation, i.e., little of the language syntax would have to be modified to eliminate clashes with "TINMAN" requirements.

TACPOL utilizes a fairly common set of constructs (in most cases). The language itself is free of machine dependent operations, and TACPOL programs could potentially be moved to a variety of machines. There are, however, a considerable set of assumptions about the compilation and target machines which may make such transportability difficult. (These include byte and word sizes, numeric precision, and an ASCII character set orientation.)

The conventions used for declaration and designation of complex data structures (e.g., cells) are difficult, not always perfectly consistent, and non-standard. It appears to be fairly easy to make a programming mistake in this regard.

TACPOL could be modified to meet most of the "TINMAN" requirements, but this would mean adding considerable syntax. Most such modifications would be straightforward, such as expanding numeric types, permitting division with remainder, adding "NOR", supporting address pointers, and changing the symbol used for relational equality. Others could be made with fairly easy language modifications, but may create serious translator and run-time difficulties, such as description of object environment, changing the iterative and conditional structure syntax rule (including making ELSE mandatory), defining recursive procedure calls, adding the syntax to support parallel processing, and tightening up currently permitted defaults. Finally, there are some changes which may require considerable new syntax and may also create significant translator and run-time costs, such as permitting user defined types, operators, and power sets of ennumeration, expanding exception handling, and providing I/O device definitions.

# EVALUATION OF

# CMS-2

Prepared by

RLG Associates, Inc.
11250 Roger Bacon Drive
Reston, Virginia 22090

November 18, 1976

This report presents an evaluation of the CMS-2 language with the requirements listed in the "TINMAN" (DOD Requirements for High Order Computer Programming Languages, "TINMAN" - 1 March, 1976, Section IV). For purposes of this comparison CMS-2 is considered to be defined by:

USER'S REFERENCE MANUAL (U) FOR COMPILER MONITER SYSTEM (CMS-2) FOR USE WITH AN/UYK-7 COMPUTER -- M-5035

CMS-2Y USER'S REFERENCE MANUAL (PRELIMINARY COPY) -- M-5049

Throughout this report, the use of the term CMS-2 denotes CMS-2Y, the version of the CMS-2 system designed for use with the AN/UYK-7 Computer.

There are 78 language requirements listed in Section 4 of the "TINMAN". This report compares CMS-2Y with each individual requirement. A summary of the degree to which the language satisfies each requirement is presented.

The introductory paragraph of each "TINMAN" requirement is included as the leading section for each requirement evaluation.

Symbols placed beside each individual requirement indicate the degree to which the language meets the requirement. The symbols and their meaning are as follows:

T - Totally meets requirement

P - Partially meets requirement

F - Does not meet requirement

U - Unknown from available documentation

# A. DATA AND TYPES

Requirement: A1

The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source program.

---p---

In general, CMS-2 is a typed language. Most types are known at compile time. However, there are a few instances where typing is not required by CMS-2. These are as follows:

Data structures can be defined to consist of machine-dependent word groups. In these cases, it is strictly up to the user to be cognizant of any data manipulation problems which may arise.

Additionally, the OVERLAY construct provides a method for violation of type checking mechanisms.

In CMS-2 formal parameters need not be explicitly typed. CMS-2 assumes a default numeric type for parameters.

Implicit variable declarations can be made in CMS-2. Again, CMS-2 assumes a default integer type for such declarations.

Type checking is performed at compile time. However, numeric data units are considered to be of the same type during assignment and evaluation of expressions; implicit conversions take place when two incompatible numeric units are manipulated. This fact coupled with the use of the OVERLAY construct provides the means for violation of type checking mechanisms. In this sense, CMS-2 does not meet the full intent of this "TINMAN" requirement.

---

Requirement: A2

The language will provide data types for integer, real

(floating point and fixed point), boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

---P---

Integer.....T
Fixed point.....T
Floating point.....T
Character string.....T
Boolean.....T
Arrays.....P
Records.....P

Arrays and Records -- CMS-2 permits both arrays (called TABLEs) and Records (also called TABLEs). CMS-2 does not however, provide the capability of using arrays and records as type generators. Thus in this sense, CMS-2 does not meet the full intent of the "TINMAN".

---

Requirement: A3

The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

---P---

There is no language element to specify the precision to be used in floating point arithmetic operations. The default precision used in all floating point operations is that of the AN/UYK-7 computer.

The cost of implementing floating point precision specification should not be too great. However, programs currently core restricted may be affected adversely.

---

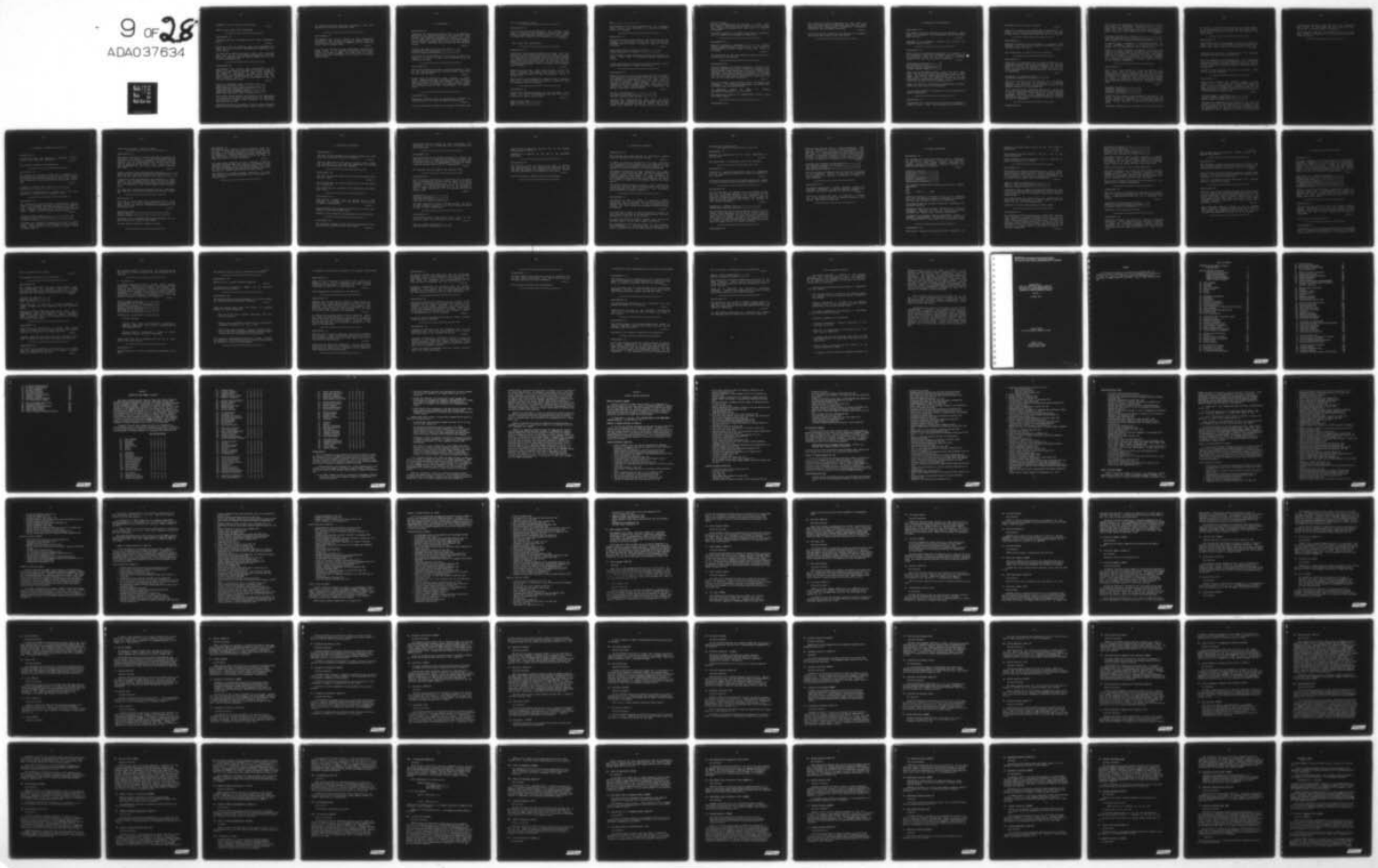
Requirement: A4

Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor

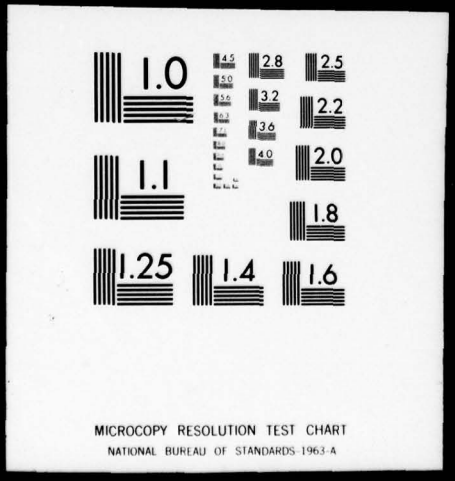
AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

9 OF 28
ADA037634



9 OF 18
ADA037634



management will be done by the compiler.

---T---

CMS-2 totally meets this requirement.

---

Requirement: A5

Character sets will be treated as any other enumeration type.

---F---

CMS-2 has a built in character type, which represents the ASCII set in an internal, implementation dependent, collating order.

The addition of EBCDIC and other widely used character sets, and the run time conversion packages to translate one set to another, could be done at modest cost to the existing implementations.

---

Requirement: A6

The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

---P---

Number of dimensions fixed at compile time.....T
Type fixed at compile time.....P
Lower subscript bound fixed at compile time....T
Subscripts from contiguous range.....T
Subscripts from enumeration type.....P
Upper subscript bound fixed at scope entry.....T

Since CMS-2 allows untyped word groups to be components of arrays, the language partially fails the requirement that the type of each array component be explicitly specified by the user.

Subscripts can only be integers in CMS-2. Since character sets are not an enumeration type (see A5 above) in CMS-2,

the language partially fails the requirement that subscripts can be from an enumerated type.

---

Requirement: A7

The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

---T---

While records (TABLEs) can have alternative structures in CMS-2, there is no explicit requirement that all components of records be named, nor that they be typed. CMS-2 allows array components to be a group of contiguous machine words.

## B. OPERATIONS

Requirement: B1

Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

---p---

Variable declaration for all data types.....T
Encapsulated type declaration.....F
Array or record declaration.....T

Encapsulated type definitons are not a part of CMS-2. All items in an array or table can be assigned using one statement in CMS-2.

---

Requirement: B2

The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

---p---

Arrays cannot be compared in their entirety in CMS-2. CMS-2 assumes some type conversions when two distinctly typed objects are compared; for example, when a fixed-point data object is compared to a floating point object, the comparison would be done algebraically in floating point.

---

Requirement: B3

Relational operations will be automatically defined for numeric data and all types defined by enumeration.

---p---

All six relational operators are included. Unordered sets

are not included in CMS-2.

---

Requirement: B4

The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

---T---

CMS-2 meets this requirement.

---

Requirement: B5

Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

---P---

There are no explicit range specifications within the CMS-2 language. The ranges are implicit in that the various types represent the data internally in 16,32 or 64 bit form.

There is only one precision for floating point numbers, and CMS-2 uses the AN/UYK-7 rounding instructions, when the rounding option is invoked.

---

Requirement: B6

The built-in boolean operators will include "AND", "OR", "NOT", and "OR". The operations "AND" and "OR" will be evaluated in short circuit mode.

---P---

Short circuit "AND".....T
Short circuit "OR".....T
NOT.....T

NOR.....F

The operation "OR" is not included within the language. Short circuit mode evaluation for scalars is defined within the language.

---

Requirement: B7

The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

---p---

Assignment between records and arrays.....T
Scalar operations on arrays.....P

Data transfer between arrays and records are permitted in CMS-2. CMS-2 makes no distinction between records and arrays.

Scalar operations are permitted only on variables, and on individual elements of arrays and records.

---

Requirement: B8

There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

---p---

Explicit conversions.....F
No implicit conversions.....F
Explicit between scale factors.....T

Implicit type conversions are used within the CMS-2 language when certain options are invoked; for example, in relational expressions, when an integer and a real number are compared, the integer would be converted to

the real format.
Explicit conversions are not permitted in CMS-2. Since all numeric data units are considered to be of the same type, internal conversions take the place of explicit conversions.

The format capability on output allows object representations of numbers to be decoded as characters.

---

Requirement: B9

Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

---F---

The capability for run time exception checking can be included within the language.

---

Requirement: B10

The base language will provide operations allowing programs to interact with files, channels or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

---F---

CMS-2 is oriented toward a serial batch processing environment. Its input/output statements are insufficient to handle real time applications.

No capability exists in CMS-2 for dynamic assignment/reassignment of I/O devices.

The addition of powerful I/O capabilities to CMS-2 would be a major effort.

---

Requirement: B11

The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

---F---

Power sets are not defined in the language. This capability can be added without much difficulty.

---

## C. EXPRESSIONS AND PARAMETERS

Requirement: C1

Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

---T---

Arguments of an expression causing side effects are evaluated left-to-right.

---

Requirement: C2

Which parts of an expression constitute the operands to each operation, within that expression, should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

---P---

Unambiguous presentation.....P
Few precedence levels.....T
Require explicit parentheses.....F
No user defined levels.....T

CMS-2 has six precedence levels. There are some cases where the use of the unary minus operator leads to ambiguity. For example, -3**2 evaluates in CMS-2 to 9; but -A**2 where A has the value 3, evaluates to -9. This evaluation of exponentiation is done from right-to-left in the latter case, and is ambiguous at best.

Again, the rules for evaluation of substitution declarations are quite complex and confusing.

In most cases however, the operands to each operation are easily discernable.

---

Requirement: C3

Expressions of a given type will be permitted anywhere in source programs where both constants and references to

variables of that type are allowed.

---T---

There are no special case constraints on expressions included in the syntax of the language, except in the case of substitution declarations where the form is different.

---

Requirement: C4

Constant expressions will be allowed in programs where constants are allowed, and constant expressions will be evaluated before run time.

---T---

This requirement is fully met by the language.

---

Requirement: C5

There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types for exception handling, for parallel processes, for declaration, or for built-in operators. There will be no special operations (e.g., array structuring) applicable only to parameters.

---P---

Consistency in parameter rules.....T
No special parameter operations.....P

CMS-2 has a consistent set of rules for all parameters. However, for the sake of efficiency, the language designers have included a clause allowing the use of the machine registers to pass parameters.

This language clause, "PARAMETER DECLARATION" associates one of the user specified AN/UYK-7 machine registers with a formal parameter. This clause is highly machine-dependent and is not in full agreement with the "TINMAN" requirement that there be no special operations applicable only to parameters.

---

Requirement: C6

Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

---P---

Dimensions determined at compile time.....T
Size and subscripts can be passed.....P
Type agreement for actual and formal parameters.....P

In general there is agreement of type between formal and actual parameters. However, as pointed out in A1, all numeric data is considered to be of the same type.

There is no direct mechanism for passing of size and subscripts of arrays. However, through the use of the major index (size indicator) of simple tables (one dimensional arrays), the size of an array can be passed as a parameter. This is really a restrictive and hidden feature, and practically speaking, the language fails the intent of the "TINMAN" requirement.

---

Requirement: C7

There will be four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

---P---

Parameter constants.....P
Parameter variables.....P
Exception conditions.....T
Procedure parameters.....F

CMS-2 calls are made by value, except in the case of indirect tables, in which case the CORAD function call may be used to pass the address of a load-time allocated table. Actual parameter values are not affected by the call.

Exception conditions can be stated by the use of the "EX-

IT" clause. The "EXIT" clause allows for program control to be transferred to one of the named statement labels, which have to be within the scope of the calling procedure block.

---

Requirement: C8

Specification of the type, range, precision, dimension, scale and format of parameters will be optional on the formal side. None of them will be alterable at run time.

---F---

CMS-2 does not allow for deviation from the specified language rules for writing procedures.

It is not apparent to the reviewer how this requirement can be achieved and the requirements of A1 and A2 simultaneously achieved. Perhaps a special construct, GENERIC, to be used only by an elite tightly controlled group of programmers.

Support of this capability requires a significant change to the the CMS-2 language.

---

Requirement: C9

There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

---P---

Variable number of arguments.....P
All but constant number same type.....F
Number fixed at compile time.....T

The number of arguments for all procedures is fixed at compile time. CMS-2 does allow for the omission of unrequired parameters on any given call to a procedure. This is accomplished by the use of commas to denote missing arguments in a call. However, it is up to the called pro-

cedure to be cognizant of the fact that a few parameters are missing. In this sense, one could state that CMS-2 allows for a variable number of arguments.

The requirement that all but a constant set of the variable number of arguments be of the same type is not enforced in CMS-2.

---

## D. VARIABLES, LITERALS AND CONSTANTS

Requirement: D1

The user will have the ability to associate constant values of any type with identifiers.

---T---

The language provides this capability.

---

Requirement: D2

The language will provide a syntax and a consistent interpretation for literals of built-in data types. Numeric literals will have the same value (within the specified precision) in both programs and data (input or output).

---P---

Literals of built-in data types.....T
Consistency in value.....U

The existing documentation is unclear as to the consistency of values in both programs and data.

---

Requirement: D3

The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

---P---

Declare initial values.....P
Allocation scope initialization.....T
No default values.....T

Variables and components of tables can be set to predefined values. However, whole arrays or tables cannot be preset. Local variables (local indexes in CMS-2) cannot be preset either.

There are no default values in CMS-2.

---

Requirement: D4

The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

---P---

Numeric variable range specification mandatory.....F
Step size for fixed point numbers must be specified.....T
Range and step size not defining a new type.....T

No explicit range specifications are mandatory in CMS-2. However, the absolute maximum value a fixed point variable can take is implied by the number of bits requested in the definition. The minimum values cannot be specified.

The range for floating point numbers are not specifiable and default to the range allowed by the AN/UYK-7 computer.

---

Requirement: D5

The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.

---F---

Enumeration types.....F
Defined Types.....F
Built-in Types.....T

Variables, array components and Record components can assume values of the built-in data types.

No user defined types are allowed in CMS-2.

---

Requirement: D6
The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

---F---

The only pointer mechanism as such, available in CMS-2 is the CORAD function call which returns the address of the requested variable, table or array component. This could be used for reference and assignment purposes. The CORAD function is intended primarily for indirect tables.

The addition of a powerful pointer capability to CMS-2 will require significant modifications to the present implementations.

---

## E. DEFINITION FACILITIES

Requirement: E1

The user of the language will be able to define new data types and operations within his programs.

---F---

CMS-2 is restricted to the use of built-in data types. The definition of new types and operations as intended here are not supported by the language.

---

Requirement: E2

The "use" of defined types will be indistinguishable from built-in types.

---F---

The language does not permit definition of new data types as an operation.

This capability can be added to the language as an extension of E1.

---

Requirement: E3

Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

---P---

Local variables can be declared implicitly in CMS-2. They default to a signed-integer type.

Removal of such implicit declarations should be trivial.

---

Requirement: E4

The user will be able, within the source language, to extend existing operators to new data types.

---F---

The language does not permit new type definition. This capability can be included as part of the extension to type definition.

---

Requirement: E5

Type definitions in the source language will permit definition of both the class of data objects comprising the types and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

---F---

The language does not support user defined types.

---

Requirement: E6

The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

---P---

Enumeration.....F
Cartesian products.....T
Discriminated union.....F
Power set of enumeration type....F

The TABLE mechanism is used to define records. Its use is limited, however. Records do not define types and cannot be used as type constructors.

---

Requirement: E7

Type definitions by free union (i.e., union of nondisjoint types) and subsetting are not desired.

---F---

Does not permit free unions.....F
Does not permit subsetting.....F

Free unions are permitted with the use of the OVERLAY capabilities in CMS-2.

Subsetting is possible by the use of the SUB-TABLE mechanism.

---

Requirement: E8

When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

---F---

New types cannot be defined within the language.

---

## F. SCOPES AND LIBRARIES

Requirement: F1

The language will allow the user to distinguish between scope of allocation and scope of access.

---F---

In CMS-2, the scope of allocation of all data structures is the duration of the program, with the exception of local variables (local indexes) where the variable is accessible for the duration of the defining procedure.

The scope of access for all data structures is, again, throughout the program for data declared in system-data-designs (global definitions); for data declared in local-data-designs the scope of access is limited to those groups of procedures; for local indexes, the scope of access is limited to the procedure defining the local index.

The modifications required to allow a true local/global definition facility would be significant, since the required changes would have to allow for both allocation and deallocation of data structures.

---

Requirement: F2

The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

---F---

As stated above, access to data structures is limited by the physical location of the data declarations. The only exception to this rule is as follows:

By using the EXTDEF and EXTREF clauses data structures declared locally can be made globally accessible.

The assessment of the cost and impact of the modifications necessary to incorporate this requirement can be made in conjunction with the modifications neccessary for

implementing requirement F1.

---

Requirement: F3

The scope of identifiers will be wholly determined at compile time.

---T---

This requirement is completely met by the language.

---

Requirement: F4

A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

---T---

Applications oriented data and subroutines can be stored in common libraries which are easily accessible by CMS-2.

---

Requirement: F5

Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

---P---

Accessible at compile time.....T
Other source language routines.....P

The language does not preclude routines written in a different source language. As long as the routines exist in a source form compatible with the CMS-2 library manipulators they can reside in system libraries. There is no provision for interface checking in the language. Hence, it cannot be said whether foreign language routines can exist in object form in the system libraries.

---

Requirement: F6

Libraries and compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

---P---

Libraries and compools indistinguishable.....P
Hold anything definable in language.....T
Many levels of access.....T
Many specialized subsets.....T

CMS-2 libraries and compools can hold anything definable in the language. However, for elements being selected from a compool, the user has to state that a compool is being selected.

---

Requirement: F7

The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

---F---

CMS-2 has a limited high level I/O capability. A limited set of peripheral equipment and special hardware is supported. See comments under B10.

---

## G. CONTROL STRUCTURES

Requirement: G1

The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

---P---

Sequential control.....T
Conditional control.....T
Iteration.....T
Recursion.....F
Parallel processing.....F
Exception handling.....P
Asynchronous interrupts.....F

The language provides the following keywords for control mechanisms:
FOR
GOTO
IF . . . . THEN . . . . ELSE
VARY

Exception handling is provided by use of the abnormal exit and the ability to alter the sequence of execution overflow upon condition detection of parameters.

The language does not provide an explicit mechanism for interrupt handling.

Similarly, there is no explicit definition of parallel processing. Addition of this to the structure of the language will require major language modification.

In general, the language control mechanisms conform to "TINMAN" requirements. The addition of the other "TIN-MAN" capabilities to CMS-2 would be a non-trivial effort.

---

Requirement: G2

The source language will provide a "GOTO" operation ap-

plicable to program labels within its most local scope of definition.

---p--

CMS-2 permits the GOTO target to be any label in the system-procedure.

Rules governing the GOTO operation can be modified to preclude branches out of scope.

---

Requirement: G3

The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

---p---

Based on Boolean expression.....T
Based on type from discriminated union.....F
Based on computed choice.....T

Conditional control based on Boolean expressions is provided for by the IF-THEN-ELSE construct. Case statements and switches provide for a computed choice among labeled alternatives.

The language does not require that all alternatives be accounted for; for example, an "ELSE" clause is not mandatory for all "IF-THEN" expressions.

There are no simple mechanisms for common cases.

---

Requirement: G4

The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity, or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

---p---

Termination anywhere in loop.....T
Local control variables.....F
Entry at loop head only.....F
Efficient and clear simple cases.....T
Control value available at termination.....P

Termination within a loop is made possible by several constructs within the language. Termination can be at the head of the loop (after a fixed number of iterations or failure of a test - WHILE), at the end of the loop (test - UNTIL) and within the loop (using the GOTO).

Loops can be entered at any labeled statement.

The control variable may or may not occur within the controlled statement. The controlled variable is a word reference, i.e., either an anonymous reference or a declared word reference. The value of the controlled variable is available for declared word references.

The rules for iterative evaluation can be redone to completely meet "TINMAN" requirements.

---

Requirement: G5

Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

---F---

Recursive and nonrecursive routines.....F
Explicitly specify recursive routines.....F
No nested recursive procedures.....F

Recursion is not provided in CMS-2.

---

Requirement: G6

The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

---F---

The language does not explicitly address parallel processing requirements.

---

Requirement: G7

The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

---F---

CMS-2 does not provide a general exception handling capability. The only constructs available are abnormal exit parameters and overflow conditions.

Addition of exception control constructs to the language will be a nontrivial process.

---

Requirement: G8

There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, and which permit asynchronous hardware interrupts to be treated as any other exception situation.

---F---

This requirement appears to expand on the G6 requirements. Parallel processing capabilities can be added to the language. This addition will be costly and may tend to cloud the readability and clarity of the language.

---

## H. SYNTAX AND COMMENT CONVENTIONS

Requirement: H1

The source language will be free format with an explicit statement separator, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple, uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

---P---

Free format with statement separator.....T
Mnemonically significant identifiers.....T
Conventional forms.....P
Simple uniform grammar.....P
No special case notations.....P
No abbreviation of keywords or identifiers....T
Unambiguous grammar.....P

CMS-2 is a free-format language. There are a number of special cases used in the grammar of the language, for example, the rule for Boolean operations is quite complex. The rules for substitution expressions are different from normal usage.

---

Requirement: H2

The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms, or define new infix operator precedence.

---T---

The syntax is completely fixed.

---

Requirement: H3

The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using

the 64 character ASCII subset.

---T---

The language satisfies this requirement.

---

Requirement: H4

The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

---P---

Literals for numbers.....T
Character strings.....T
Break character.....F

Identifiers can be from one to eight characters in length. However, the first character has to be an alphabetic.

There are no break characters, per se, in CMS-2. As a consequence, there are elaborate rules as to where a space can be used, and the usage is context dependent. Break characters could be introduced into the language with minor modifications.

---

Requirement: H5

There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

---F---

Multiple lines are permitted within the language structure. It should be very simple to change this feature.

---

Requirement: H6

Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

---P---

The language defines 179 key words. The proliferation of key words is highly indicative of the complexity of the language.

---

Requirement: H7

The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

---P---

Single comment convention.....F
Distinguishable from code.....P
Introduced by one or two characters.....P
Contain any character.....T
Appear anywhere reasonable.....T
Terminate at end of line.....F
Not prohibit reformatting.....T

Comments may be expressed in three forms:

COMMENT <string>

Bracket Notes: These are specialized documentation features associated with system declarations and data declarations.

Strings quoted by apostrophes: These can appear anywhere a space can within a statement.

These three forms can be replaced with just one of these forms, with minor effort.

---

Requirement: H8

The language will not permit unmatched parentheses of any kind.

---T---

The language does not permit unmatched parentheses.

---

Requirement: H9

There will be a uniform referent notation.

---P---

All notation is consistent, except for the intrinsic functions, BIT and CHAR.

---

Requirement: H10

No language defined symbols appearing in the same context will have essentially different meanings.

---P---

There are several cases where the meaning of language symbols is context dependent.

- AND and OR are both logical connectors and bit-string operators.

- THEN is both a statement connector and a conditional statement connector, i.e., IF ... THEN.

- EQUALS has three meanings - numeric constant EQUALS, the difference in addresses between allocated identifiers and the allocation of an identifier.

This leads to considerable ambiguity in usage. Changes to eliminate this confusion would require changing both the syntax as well as the translators.

---

## I. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

Requirement: I1

There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

---T---

This requirement is satisfied by the language.

---

Requirement: I2

Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

---P---

In general the language does not permit defaults. However, there are some exceptions. For example, the mode of packing of TABLE fields can be selected by the programmer or he can let the compiler manage it optimally. Reentrant procedures have to be explicitly specified or the mode is non-reentrant.

---

Requirement: I3

The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

---F---

CMS-2 does not allow this capability. The only available compile time feature is the ability to selectively compile pieces of source code. Addition of this capability would require major modifications.

---

Requirement: I4

The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the condition will be evaluated at compile time and only the selected path will be compiled.

---F---

Selective compilation is possible using the CSWITCH mechanism. This allows for compilation only when the CSWITCH "flag variable" is set on. This capability does not meet the full intent of the "TINMAN" requirement.

---

Requirement: I5

The source language will contain a simple, clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

---F---

As far as can be determined, the base or kernel constitutes the entire language.

---

Requirement: I6

Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

---P---

Available documentation describing compiler limits is precise in some areas and vague in others. For example, array dimensions, identifier lengths, and nesting limits are explicitly defined. The number of identifiers allowable is not explicitly defined.

Limits are probably dependent upon the machine environment in at least some cases.

---

Requirement: I7

Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

---T---

The language satisfies this requirement.

---

## J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

Requirement: J1

The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

---U---

It is not clear from the documentation as to how the code generators function. Also, the evaluator cannot determine whether run-time support routines are used.

---

Requirement: J2

Any optimizations performed by the translator will not change the effect of the program.

---T---

Insofar as it is known to the evaluator, optimization done by the CMS-2 compiler does not change the effect of the program.

---

Requirement: J3

The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

---T---

The DIRECT CODE construct provides this capability.

---

Requirement: J4

It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation

will be optimal as determined by the translator.

---P---

Specify object presentation.....P
Specify time/space tradeoff.....P

This capability is somewhat available through use of the TABLE construct. Field orders, field widths, bit patterns and field structures can all be specified within a table structure.

There is no mechanism for explicitly specifying tradeoffs. However, the translators do in fact optimize data manipulations for the object computer.

---

Requirement: J5

The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and closed routine of the same description will have identical semantics.

---F---

All user defined procedures and functions are closed. The language could be modified to add this capability.

## CMS-2 Evaluation Summary

This section presents a summary of the findings resulting from this evaluation of CMS-2. The strengths and weaknesses of the language are noted. The feasibility of modifying the language to fully meet the "TINMAN" requirements is discussed.

CMS-2 has the following strong points in comparison with the "TINMAN".

- Well Defined Syntax -- syntax of the language cannot be changed; no keyword abbreviations; language is free format.

- Binding Capability -- programs may be compiled separately and linked at execution time; COMPOOLS can be wed to hold intermediate output.

The major weaknesses of the language in comparison to the "TINMAN" are as follows:

- Parallel processing not supported.

- A general exceptional handling capability is not available in CMS-2.

- Powerful I/O capabilities, as required by the "TINMAN", are not supported.

- No facilities for user definable data types, and the associated ability to define new operations for such data types.

- Strong typing, as required by the "TINMAN", is not rigidly enforced in CMS-2.

In general, CMS-2 provides an adequate framework for

problem solving. Since the base, or kernel, of the language constitutes the entire language, it is a fairly complex one. For example, the documentation lists 179 key-words. The language allows some machine-dependent operations; such operations include register use, word size and internal data representations. Certainly these constructs can be removed, but at some expense. As is the case with other languages implemented on a variety of machines, a number of implementation dependencies have crept into CMS-2. Applications programs taking advantage of such installation dependent features may have to be modified to operate on other machines.

CMS-2 programs tend to be hard to read, due to the use of a limited character set (026 codes), as well as the use of three distinct comment conventions. Since maintainability is a major DOD goal, readability of programs is highly desirable.

Modification of CMS-2 to fully meet the DOD "TINMAN" requirements is possible. Some features, such as an extended pointer capability and automatic range checking, can be easily included. Other features, such as expanded array and record handling capabilities, recursive control structures, powerful I/O capabilities, parallel processing and strong typing could prove to be costly, both in terms of time and money. In fact, the modifications necessary to incorporate the latter capabilities would destroy much of the flavor of CMS-2, as it is known today.

**SofTech, The Software Technology Company**
460 Totten Pond Road, Waltham, Massachusetts 02154 617-890-6900

# EVALUATION OF ALGOL 68, JOVIAL J3B, PASCAL, SIMULA 67, AND TACPOL VS. TINMAN REQUIREMENTS FOR A COMMON HIGH ORDER PROGRAMMING LANGUAGE

1021-14

October 1976

Prepared under
U.S. Army Contract DAAB07-75-C-0373

SofTech, Inc.
460 Totten Pond Road
Waltham, Mass. 02154

170 pages

(12)

## PREFACE

The evaluations presented in this report were performed under the direction of John B. Goodenough, who also was responsible for the evaluation of JOVIAL J3B. Clement L. McGowan was primarily responsible for the evaluation of PASCAL, SIMULA 67, and TACPOL. John R. Kelly evaluated ALGOL 68.

TABLE OF CONTENTS


| | | |
|---|---|---|
| INTRODUCTION AND SUMMARY OF RESULTS | | 1 |
| | Recommendations | 3 |
| DETAILED LANGUAGE EVALUATIONS | | 6 |
| | ALGOL 68 Evaluation Summary | 6 |
| | J3B Evaluation Summary | 8 |
| | PASCAL Evaluation Summary | 11 |
| | SIMULA 67 Evaluation Summary | 14 |
| | TACPOL Evaluation Summary | 17 |
| A1. | Typed Language | 20 |
| A2. | Data Types | 21 |
| A3. | Precision | 23 |
| A4. | Fixed Point Numbers | 24 |
| A5. | Character Data | 26 |
| A6. | Arrays | 27 |
| A7. | Records | 29 |
| B1. | Assignment and Reference | 30 |
| B2. | Equivalence | 32 |
| B3. | Relationals | 33 |
| B4. | Arithmetic Operations | 35 |
| B5. | Truncation and Rounding | 36 |
| B6. | Boolean Operations | 37 |
| B7. | Scalar Operations on Composite Data Structures | 39 |
| B8. | Type Conversion | 40 |
| B9. | Changes in Numeric Representations | 43 |
| B10. | I/O Operations | 44 |
| B11. | Power Set Operations | 47 |
| C1. | Side Effects (Due to Evaluation Order) | 48 |
| C2. | Operand Structure | 49 |
| C3. | Expressions Permitted | 51 |
| C4. | Constant Expressions | 52 |
| C5. | Consistent Parameter Rules | 54 |
| C6. | Type Agreement in Parameters | 56 |
| C7. | Formal Parameter Kinds | 59 |
| C8. | Formal Parameter Specifications | 60 |
| C9. | Variable Numbers of Parameters | 62 |
| D1. | Constant Value Identifiers | 63 |
| D2. | Literals | 64 |
| D3. | Initial Values of Variables | 65 |
| D4. | Numerical Range and Step Size | 67 |
| D5. | Variable Types | 68 |
| D6. | Pointer Variables | 70 |
| E1. | User Definitions Possible | 72 |
| E2. | Consistent Use of Types | 73 |
| E3. | No Default Declarations | 74 |
| E4. | Can Extend Existing Operators | 75 |

E5. Type Definitions 76
E6. Data Defining Mechanisms 78
E7. No Free Union or Subset Types 80
E8. Type Initialization 81

F1. Separate Allocation and Access 82
F2. Limiting Access Scope 84
F3. Compile Time Scope Determination 86
F4. Libraries Available 88
F5. Library Contents 89
F6. Libraries and Compools Indistinguishable 90
F7. Standard Library Definitions for Machine-Dependent Interfaces 92

G1. Kinds of Control Structures 94
G2. The GOTO 95
G3. Conditional Control 96
G4. Iterative Control 98
G5. Routines 99
G6. Parallel Processing 101
G7. Exception Handling 102
G8. Synchronization and Real Time 104

H1. General Syntax Characteristics 105
H2. No Syntax Extensions 107
H3. Source Character Set 108
H4. Identifiers and Literals 109
H5. Lexical Units and Lines 111
H6. Key Words 112
H7. Comment Conventions 113
H8. Unmatched Parentheses 115
H9. Uniform Referent Notation 116
H10. Consistency of Meaning 119

I1. No Defaults in Program Logic 121
I2. Object Representation Specifications Optional 122
I3. Compile Time Variables 124
I4. Conditional Compilation 125
I5. Simple Base Language 127
I6. Translator Restrictions 129
I7. Object Machine Restrictions 131

J1. Efficient Object Code 132
J2. Safe and Effective Optimization Possible 134
J3. Machine Language Insertions 136
J4. Object Representation Specifications 137
J5. Open and Closed Routine Calls 139

K1. Operating System Not Required 140
K2. Program Assembly 142
K3. Software Development Tools 143
K4. Translator Options 144
K5. Assertions and Other Optional Specifications 146

| | | |
|---|---|---|
| L1. | No Superset Implementations | 147 |
| L2. | No Subset Implementations | 148 |
| L3. | Low Cost Translation | 150 |
| L4. | Many Object Machines | 151 |
| L5. | Self-Hosting Possible | 152 |
| L6. | Translator Checking Required | 153 |
| L7. | Diagnostic Messages | 154 |
| L8. | Translator Internal Structure | 156 |
| L9. | Self-Implementable Language | 157 |
| M1. | Existing Language Features Only | 158 |
| M2. | Unambiguous Definition | 159 |
| M3. | Language Documentation Required | 161 |
| M4. | Control Agent Required | 162 |
| M5. | Support Agent Required | 163 |
| M6. | Library Standards and Support Required | 164 |

## SECTION 1

## INTRODUCTION AND SUMMARY OF RESULTS

This report evaluates how well ALGOL 68, JOVIAL J3B, PASCAL, SIMULA 67, and TACPOL satisfy the Department of Defense requirements for high order computer programming languages, as described in "A Common Programming Language for the Department of Defense -- Background and Technical Requirements," Institute for Defense Analyses, Arlington, Va., Paper P-1191, June 1976, known informally as the "TINMAN." The body of this evaluation report consists of stating for each TINMAN requirement (e.g., "A6. Arrays") the degree to which each language satisfies the requirement (e.g., "Not Satisfied"), together with a justification for this evaluation, keyed to language reference documents. Bibliographic references to these documents are found at the beginning of Section 2, where each language's deficiencies are summarized. The remainder of Section 2 consists of the detailed language evaluations. Each evaluation begins by quoting relevant parts of the TINMAN requirement, and then each language is evaluated separately with respect to that requirement.

A summary of how well each language satisfies each requirement is presented below, where S indicates the requirement is satisfied, P indicates it is partially satisfied, N indicates it is not satisfied, and U indicates it was not possible to determine whether the requirement was satisfied or not.

| | | ALG | J3B | PAS | SIM | TAC |
|---|---|---|---|---|---|---|
| A1. | Typed Language | S | P | P | P | P |
| A2. | Data Types | P | P | P | P | P |
| A3. | Precision | N | N | N | N | N |
| A4. | Fixed Point | N | N | N | N | P |
| A5. | Character Data | N | N | N | N | N |
| A6. | Arrays | P | P | P | P | P |
| A7. | Records | P | P | P | P | N |
| B1. | Assignment | P | P | P | P | P |
| B2. | Equivalence | P | P | P | P | N |
| B3. | Relationals | P | P | P | P | P |
| B4. | Arithmetic Operations | P | P | P | S | P |
| B5. | Truncation/Rounding | P | P | U | P | N |
| B6. | Boolean Operations | P | P | N | P | P |
| B7. | Scalar Operations | P | P | P | N | N |
| B8. | Type Conversion | N | P | N | N | P |
| B9. | Range Exceptions | N | P | P | N | S |
| B10. | I/O Operations | S | N | P | S | P |
| B11. | Power Set Operation | N | N | P | N | N |
| C1. | Side Effects | N | N | N | N | N |
| C2. | Operand Structure | N | P | P | N | P |
| C3. | Expressions Permitted | S | S | S | S | U |
| C4. | Constant Expressions | N | P | N | N | N |

| | | | | | | |
|---|---|---|---|---|---|---|
| C5. | Parameter Rules | P | P | P | P | N |
| C6. | Parameter Agreement | P | P | P | P | S |
| C7. | Parameter Kinds | P | P | N | N | P |
| C8. | Parameter Specifications | N | N | N | P | N |
| C9. | Numbers of Parameters | S | N | N | N | N |
| D1. | Constant Value Identifiers | S | S | S | N | S |
| D2. | Numeric Literals | P | P | P | P | P |
| D3. | Initialization | P | P | N | N | N |
| D4. | Ranges and Step Sizes | N | N | N | N | P |
| D5. | Variable Types | P | P | S | P | P |
| D6. | Pointer Variables | S | P | N | S | N |
| E1. | User Definitions | S | N | P | P | N |
| E2. | Consistent Use | S | N | N | P | N |
| E3. | No Default Declarations | S | S | S | P | P |
| E4. | Extend Existing Operators | P | N | N | N | N |
| E5. | Type Definitions | N | P | P | P | N |
| E6. | Data Definitions | P | P | S | P | N |
| E7. | No Free Union | S | N | N | S | N |
| E8. | Type Initialization | N | N | N | P | N |
| F1. | Allocation and Access | P | P | P | P | P |
| F2. | Limit Access Scope | P | P | N | P | P |
| F3. | Static Scope Determination | S | S | S | S | S |
| F4. | Libraries Available | N | N | N | P | N |
| F5. | Library Contents | N | P | N | N | P |
| F6. | Libraries and Compools | N | N | N | P | P |
| F7. | Machine-Dependent Interfaces | S | N | S | S | S |
| G1. | Control Structures | P | P | P | N | P |
| G2. | The GOTO | N | P | N | N | N |
| G3. | Conditional Control | P | N | P | N | N |
| G4. | Iterative Control | P | N | N | N | P |
| G5. | Routines | P | S | P | P | U |
| G6. | Parallel Processing | P | N | N | N | N |
| G7. | Exception Handling | N | N | N | N | P |
| G8. | Synchroniztion | N | N | N | N | N |
| H1. | Syntax Characteristics | S | P | P | S | P |
| H2. | No Syntax Extensions | N | S | S | S | S |
| H3. | Source Character Set | S | S | P | N | S |
| H4. | Identifiers and Literals | P | P | P | P | P |
| H5. | Lexical Units | N | N | N | N | N |
| H6. | Key Words | S | P | S | S | U |
| H7. | Comment Conventions | P | P | P | N | P |
| H8. | Unmatched Parentheses | S | S | S | S | S |
| H9. | Uniform Referent Notation | P | P | N | N | S |
| H10. | Consistency of Meaning | P | N | P | N | N |
| I1. | Defaults in Program Logic | S | N | N | N | N |
| I2. | Object Representation | P | S | P | N | P |

| | | | | | | |
|---|---|---|---|---|---|---|
| I3. | Compile Time Variables | P | N | N | N | N |
| I4. | Conditional Compilation | N | S | N | N | N |
| I5. | Simple Base Language | P | P | S | P | N |
| I6. | Translator Restrictions | N | P | N | N | P |
| I7. | Object Machine Restrictions | P | N | P | P | S |
| J1. | Efficient Object Code | N | P | P | N | P |
| J2. | Optimizations Possible | P | P | P | P | N |
| J3. | Machine Language | N | N | N | N | P |
| J4. | Object Representations | N | P | N | N | P |
| J5. | Open/Closed Routines | N | S | N | N | N |
| K1. | Operating System | P | S | S | P | S |
| K2. | Program Assembly | N | S | N | P | S |
| K3. | Software Tools | U | U | U | U | U |
| K4. | Translator Options | U | U | U | U | P |
| K5. | Assertions | S | N | N | N | N |
| L1. | Supersets | N | N | S | N | N |
| L2. | Subsets | N | N | S | S | N |
| L3. | Low-Cost Translation | P | S | S | P | S |
| L4. | Many Object Machines | S | S | S | S | N |
| L5. | Self-Hosting | U | S | S | P | S |
| L6. | Translator Checking | S | S | U | P | S |
| L7. | Diagnostic Messages | N | P | N | N | N |
| L8. | Translator Structure | S | S | S | S | S |
| L9. | Self-Implementable | S | P | S | P | P |
| M1. | Existing Features | S | S | S | S | S |
| M2. | Unambiguous Definition | S | P | P | P | N |
| M3. | Language Documentation | S | P | S | S | N |
| M4. | Control Agent | N | N | N | N | N |
| M5. | Support Agent | N | N | N | N | N |
| M6. | Library Standards | N | N | N | N | N |

Recommendations

One of the purposes of this evaluation effort was to determine whether any language satisfies the TINMAN requirements sufficiently that it can be used unchanged (or with relatively minor modifications) as the DoD Common Language for embedded computer applications. None of the languages we have evaluated, however, requires only minor changes to satisfy these requirements, and so none is suitable for immediate adoption by the DoD.

A second objective of the evaluation was to select languages which could usefully serve as the basis for developing a common language. Among the reasons for choosing a basis language (and modifying it, perhaps extensively) are:

1) the common language can draw on a population of programmers familiar with the use of the basis language, assuming the changes made to it are not too drastic;

2) similarly, compilers and other tools developed for the basis language can be reused or modified if the changes required are not too extensive;

3) perhaps more important, by starting with a basis language whose deficiencies and strengths are understood, modifications can be guided by the desire to eliminate the language's deficiencies while preserving its strengths; this can help to focus language design issues;

4) design effort can be concentrated on the more difficult design issues, since choosing a basis language resolves many trivial issues immediately, and provides an orientation that can guide the resolution of other issues.

Keeping these goals in mind, a suitable basis language should satisfy at least the following criteria:

- it should have a well specified language definition (so it is clear what the basis language is);

- the additions and deletions required to satisfy the TINMAN requirements should be relatively independent, i.e., one should not have to delete a feature to satisfy a requirement and then add a similar feature to satisfy the same requirement; such modifications vitiate the value of building from the selected basis language;

- experience in using the language, especially for applications similar to embedded computer applications, is important in understanding the strengths and weaknesses of a basis language relevant to DoD applications;

- the features of a basis language that satisfy TINMAN requirements should not be a subset of another language's features, i.e., if two languages are both otherwise acceptable basis languages, one should select that language having the greatest number of acceptable features, since the experience gained in using a larger number of features is of greater benefit in guiding the modification effort.

Of the languages evaluated, only PASCAL and ALGOL 68 appear suitable as a basis for creating a language satisfying all the TINMAN requirements. PASCAL is suitable essentially because relatively few features need to be deleted from the language. On the other hand, so many features need to be added that the final language may bear little resemblance to PASCAL. In particular, maintaining PASCAL's simplicity and consistency will be difficult, but attempting to do so will in itself be a useful constraint on the design effort.

ALGOL 68 is suitable as a basis for modification primarily because relatively few features need to be added to it, and the features that need to be added would need to be added to virtually any other language selected as a design base. Most of the ALGOL 68 modifications are deletions and

simplifications. Experience has been gained in England in the use of ALGOL 68 for systems implementation. An advantage of using ALGOL 68 as a design base is its consistency -- its use of a few linguistic concepts uniformly combined. Another advantage, from the point of view of modification, is its syntactic and semantic description method, which helps to enforce the consistency and uniformity that is typical of the language. Modifications to ALGOL 68 can be described in terms of this syntax, and the ramifications of deleting or modifying some feature can be more readily traced out due to the specification method. This may well prove productive in bringing modification problems to a design team's attention. Since it is the interaction among language features that makes language design difficult, having a formal basis for tracking these interactions should prove productive.

JOVIAL J3B is unsuitable as a basis for modification primarily because of its numerous arbitrary restrictions and inconsistencies (see, for example, H1 and C5). These would all have to be removed and resolved before J3B could serve as a suitable basis, and this would not be a productive use of a design team's time.

TACPOL is unsuitable as a basis for modification primarily because its acceptable features are a (rather small) subset of the acceptable features of other languages, e.g., PASCAL.

SIMULA 67 is unsuitable primarily because the class concept, which is what distinguishes SIMULA from other languages, is unsuitable for embedded computer applications, and if this construct is deleted from the language there is no longer any virtue in using SIMULA as a basis language rather than ALGOL 60. The difficulty with the class concept is primarily that it supports record data structures only indirectly. Looking a a SIMULA class definition, it is not easy to see what class "attributes" correspond to fields in an underlying record structure, even though identification of these fields is necessary when the object representation of a record structure is to be specified explicitly by a programmer (TINMAN requirements I2 and J4). The fact that SIMULA classes abstract away from the physical structuring of a record is a virtue in its intended application area, but for embedded computer applications, where the physical structuring of data tends to be very important, this abstraction will tend to make a SIMULA-based language difficult and unwieldy to to use. Even so, it is clear that much can be learned from the SIMULA experience in deciding how to adapt other languages to meet the TINMAN requirements.

# SECTION 2

# DETAILED LANGUAGE EVALUATIONS

## ALGOL 68 Evaluation Summary

SofTech's evaluation of ALGOL 68 with respect to the Department of Defense requirements for high order computer programming languages ("TINMAN," June 1976) is summarized here. Detailed evaluations of ALGOL 68 with respect to each TINMAN requirement are contained in separate sections of this report. These sections begin with an overall assessment of the extent to which the requirement is satisfied, followed by a justification of this assessment referencing sections of the ALGOL 68 defining document:

van Wijngaarden, A. et. al. (eds.) Revised Report on the Algorithmic Language ALGOL 68. Springer-Verlag, 1976.

## Summary of Changes Required for ALGOL 68

We list here the additions and deletions that must be made to ALGOL 68 to bring it into conformance with the TINMAN requirements. An addition is considered major (and marked with an asterisk) when it potentially interacts with many language features or when it significantly changes the character of the language and its use. Each addition or deletion is cross-referenced to a TINMAN requirement justifying the modification. A more detailed description of the nature of the required change will be found in the discussion of the referenced requirement.

## Required Additions (ALGOL 68)

* change arithmetic types from implicit implementation dependent precision (via long) to just int and float with precision specified separately (A2, A3, A4, D4)
. problem-oriented floating point precision specification (A3)
. global default floating point precision specification (A3)
. "exact" fixed point real arithmetic, with specifiable precision (A4)
* specifiable character sets (A5)
. arrays indexable by enumeration types (A6)
* programmer control over discrimination of alternative record structures without having free union (A7)
* assignment and access definable for new types (B1)
. extend built-in equivalence definition to apply to all built-in atomic types (B2)
. unordered enumeration types (B3)
. define dyadic arithmetic operators for operands of mixed precision (B4)
. define rounding/truncation rules for real arithmetic (B5)
. short circuit evaluation of AND and OR operators (B6)
* define scalar operators over conformable arrays (B7)
* run time exception handling ability for range errors (B9)
. powerset of enumeration type, with operators (B11, E6)

- left-to-right evaluation order for operands, subscripts, and subroutine parameters (C1)
- require parentheses at same precedence level for non-associative operators (C2)
- require constant expressions to be evaluated at compile time (C4)
- change integer case clauses to use explicitly labeled alternatives (G3)
* exception handling, including overflow, range, and subscript errors (C7, B9, D4, G7)
- generic procedures (C8)
- require program and data numeric literals to have the same value (D2)
- run time uninitialized variable test (D3)
- range declarations (D4)
- enumeration types (D5, E6)
* group applicable operators with a type definition (E5)
* associate initialization, finalization, allocation, and deallocation actions with programmer defined types (E8)
* limit access to separately defined structures (F2)
* application-oriented library mechanism (F4, F5, F6)
- iterative control with termination condition occurring anywhere in loop body (G3, G4)
* asynchronous interrupt handling (G8)
* delay until time (real or simulated) or situation occurs (G8)
- real time clock access (G8)
- separate quoting of each line of long character string literals (H4)
- modify field selection syntax (H9)
- mechanisms for specifying object representation (I2, J4)
- extend set of environmental enquiries (I3)
- conditional compilation (I4)
- define translator restrictions/limits (I6)
- require that translators not build in object machine dependent language restrictions (I7)
* programmer control over movement between main and backing store (J1)
- machine language insertions (J3)
- associate identifiers with object machine addresses (J4)
- open and closed routine calls (J5)
* mechanism for integrating separately written modules (K2)
- software development tools (K3)
- translator options (K9)
- prohibit supersets and subsets (L2, L3)
- provide suggested error and warning messages (L7)
- DoD designated control agent, support agent, and library support (M4, M5, M6)

Required Deletions (ALGOL 68)

- string data type as built-in type (A2, I5)
- procedure variables (A2)
- bits type (A2)
- bytes type (A2)
- non-constant lower bounds for arrays (A6)
- flexible arrays (A6)
- widening coercion (implicit integer to real conversion) (B8, C6)

- formatted I/O (B10)
- priority (operator precedence) declarations (C2, H2)
- array substructuring (trimmers) as an operation with its own special syntax (C5)
- automatic inheritance by new types of the properties and operators of the underlying representation types (B7, E4, E5)
- dynamic storage allocation (F1)
- jumps from inner to outer lexical scope (G2)
- optional else and out clauses (G3)
- procedure definition within the bodies of recursive procedures (G5)
- arbitrary nesting of procedures and parallel clauses (G6)
- lexical units (including comments) continued across end of line (H5, H7)
- square brackets for array subscripting (H9)
- left hand side function calls (H10)
- alternate notations for conditional and case clauses (I5)
- conditional and case expressions (I5)
- valued assignment statement (I5)
- statements preceding declarations within a lexical scope (L3)

## J3B Evaluation Summary

SofTech's evaluation of JOVIAL J3B with respect to the Department of Defense requirements for high order computer programming languages ("TINMAN", June 1976) is summarized here. Detailed evaluations of J3B with respect to each TINMAN reuirement are contained in separate sections of this report. These sections begin with an overall assessment of the extent to which the requirement is satisfied, followed by a justification of this assessment referencing sections of the J3B defining document:

> JOVIAL/J3B Extension 2 Language Specification. SofTech, Inc., Waltham, Mass., Document 2044-4.2, July 1975.

(A later version of this specification, dated September 1976, contains only minor changes that do not affect the evaluations reported here.)

## Summary of Changes Required for J3B

We list here additions and deletions that must be made to JOVIAL J3B to bring it into conformance with the TINMAN requirements. An addition is considered major (and is marked with an asterisk) when it potentially interacts with many language features or when it significantly changes the character of the language and its use. Each addition or deletion is cross-referenced to a TINMAN requirement justifying the modification. A more detailed description of the nature of the required change will be found in the discussion of J3B with respect to the referenced TINMAN requirement.

## Required Additions (J3B)

- require explicit discrimination among record variants with programmer control over how the discrimination condition is tested (A1, A7, E7, G3)

- Boolean data type (A2)
- problem-oriented floating point precision specification (A3)
- global default floating point precision specification (A3)
- exact fixed point data type (and modified scaling rules), with specifiable precision (A4, D4)
- ability to define new character sets (A5)
- enumeration data type (A5, E6), with and without ordering (B3)
- programmer-specified lower array bound (A6)
- arrays indexable by enumeration types (A6)
- permit upper subscript array (and table) bounds to be determined on entry to allocation scope (A6)
* assignment and access functions definable for new types (B1)
- return arrays and records as values of functions (B1, E1, E2)
- floating point equivalence defined for minimum (rather than maximum) precision of operands (B2)
* programmer-specified definition of equivalence operation (B2)
- unordered enumeration types (B3)
- relational comparisons defined for character strings (B3)
- fixed point exponentiation operator (B4)
- remainder (modulus) operator (B4)
- rounding required in floating point computations (B5)
- explicit ROUND operator available for fixed and floating point data (B5)
- require short circuit evaluation of Boolean operators (B6)
- generalized array and record assignment (B7)
* ability to apply scalar operations to conformable arrays and records (B7)
- explicit conversion operators from and to floating point representations (B8)
- explicit conversion operators from and to character string representations of numeric data (B8, D2)
* run time exception handling for range errors (B9)
* input/output operations (B10)
- power set data type and operations (B11, E6)
- left to right evaluation order for operands, subscripts, and subroutine arguments (C1)
- explicit requirement for parentheses within same precedence level for non-associative operators (C2)
- extend INTR, BIT, and BYTE operators to accept expressions and constants, if these functions are not deleted from the language (C3)
- extend constant expression computations to INTR, BYTE, BIT, SHIFTR, SHIFTL, relational expressions, and exponentiation, for those operators retained in the language (C4)
- evaluate constant expressions with target machine range and precision (C4)
- change BIT and BYTE notation to reflect potential modifiability of argument (C5)
* parameterized type definitions (C5)
- permit formal integer parameters to be declared without explicit size (C5)
- permit different array and table bounds for actual subroutine parameters (C6)
- ability to pass subroutines as parameters (C7)

* exception handling control structures (C7, G7)
    * range exception (B9)
    * overflow exception (B9)
    * subscript range exception (D4)
* generic subroutines (C8)
* variable number of parameters (C9)
* provide for input of literals (D2)
. initialization of data local to a subroutine (D3)
. initialization of typed tables (D3)
. run time testing option for accessing uninitialized variables (D3)
. range declarations (D4)
. permit arrays as components of records (D5, E6)
. enumerations types (D5, E6)
* prohibit parameter assignments to variables whose lifetime is longer than that of the object pointed to (D6)
* ability to define new infix and prefix operators (E1)
. permit arrays of records to be accessed with subscripts for user-defined record types (E2)
* ability to extend existing operations to new data types (E4)
* explicit support for data abstractions (E5)
* ability to define initialization and finalization procedures associated with variables of a user-defined type (E8)
. augment ability to define scope of allocation (F1)
. provide directive for redefining names of reusable program components (F2)
* hierarchies of COMPOOL scopes (F2, F6)
* extension libraries (F4)
* permit subroutine definitions in COMPOOLs (F5)
* ability to restrict COMPOOL references to particular programs or subsystems (F6)
. provide case statement (G3)
* equivalent of Zahn's device (G3, G7)
. provide new iterative control structure (G4)
* parallel control paths, real time clocks, asynchronous hardware interrupt language features (G6, G8)
. permit labeled END statements (H1)
. provide a break character for literals (H4)
. require separate quoting of long string literals (H4)
* uniform referent notation (H9)
. provide a new assignment symbol (H10)
. provide environmental enquiries (I3)
. provide for conditional compilation of declarations (I4)
. specify translator restrictions (I6)
* programmer control over movement between main and backing store (J1)
. provide encapsulated assembly code, including provisions for describing register usage within assembly code encapsulations (J3)
. provide ability to associate source language identifiers with machine addresses (J4)
* separate description of data representation from logical properties (J4)
. provide special comment brackets for assertions, etc. (K5)

Required Deletions (J3B)

- fractional fixed point data type (A4)
- character and bit string data types (A2, B11, I5)
- floating point equivalence definition (maximum instead of minimum precision) (B2)
- relational comparisons on pointer data (B3)
- implicit truncation in fixed and floating point arithmetic and assignments (instead of rounding) (B5)
- implicit conversion from real to integer in assignments (B8)
- untyped table data type and its use in assignment and parameter passing (B7, B8, C6)
- implicit conversion from integer to real in assignments and arithmetic operations (B8)
- implicit lengthening and shortening of character and bit strings in assignments and comparisons (B8)
- unsigned integer data type (B9)
- range matching between formal and actual parameters (B9)
- bit string data type as opposed to power set data type (B11)
- evaluation of constant expressions with host machine range and precision (C4)
- untyped pointer parameters (C6)
- default variable initialization (D3)
- distinction between tables and arrays (D5)
- overlay capability (E7)
- delete COPY capability (and extend COMPOOLS to include COPY) (F6)
- SWITCH statements (G2)
- IF-THEN without an ELSE statement (G3)
- FOR statement (G4)
- eight character rule for identifiers (H1)
- do not permit lexical units to continue across line boundaries (H5)
- do not reserve ABS, BIT, BYTE, ENTRY, FIX, INTGR, INTR, LBOUND, NENT, POINT, SCALE, SHIFTL, SHIFTR, and UNBOUND (H6)
- delete quote comment form, or change use of " in macro definitions and invocations to use % or some other delimiter, or delete macros from the language (H7)
- do not permit comments to extend across line boundaries (H7)
- substring assignment (H10)
* implementation dependencies (I1)
- character string data type as a special type rather than as an array of characters (I5)
- macro definitions and invocations (I5)
- object machine restrictions (see I7)
- provision for implementation dependent built-in functions (L1)
- require fixed point implementation (L2)

PASCAL Evaluation Summary

SofTech's evaluation of PASCAL with respect to the Department of Defense requirements for high order computer programming languages ("TINMAN, June 1976) is summarized here. Detailed evaluations of PASCAL with respect to each

TINMAN requirement are contained in separate sections of this report. These sections begin with an overall assessment of the extent to which the requirement is satisfied, followed by a justification of this assessment referencing sections of the PASCAL defining document and other pertinent publications. The following documents are cited throughout the PASCAL evaluation discussions (the citation format used in these discussions is specified below in square brackets for each document):

K. Jensen and N. Wirth, PASCAL User Manual and Report, Springer-Verlag, 1974 [p. xx].

N. Wirth, "An Assessment of the Programming Language PASCAL," IEEE Trans. on Software Engineering, 1, 192-198, (1975) [Wirth, p. xx].

N. Wirth, "The Design of a PASCAL Compiler," Software -- Practice and Experience, 1, 309-333, (1971) [Wirth, SPE, p. xx].

Other references to the PASCAL Newsletter and to two formal definitions of the language are given in full citation form where they appear.

The principal reference is the User Manual and Report. Pages 133-168 comprise the revised Report defining standard PASCAL. (Pages 3-87, 105-118 are the portions of the User Manual explaining standard PASCAL, and pages 88-104, 119-125 deal with PASCAL 6000.) Standard PASCAL is to be supported by all implementations to provide program portability. Standard PASCAL is the language evaluated and called PASCAL. The language PASCAL 6000 is a particular implementation for the Control Data 6000 series of computers. It is sometimes cited to illuminate the relationship between defining and implementing a language feature.

Summary of Changes Required for PASCAL

We list here the additions and deletions that must be made to PASCAL to bring it into conformance with the TINMAN requirements. An addition is considered major (and marked with an asterisk) when it potentially interacts with many language features or when it significantly changes the character of the language and its use. Each addition or deletion is cross-referenced to a TINMAN requirement justifying the modification. A more detailed description of the nature of the required change will be found in the discussion of the referenced requirement.

Required Additions (PASCAL)

* require explicit discrimination among record variants, with complete programmer control over how the discrimination condition is tested (A1, A7, G3)
. fixed point real arithmetic with specifiable precision and step size (A2, A4, D4)
. problem-oriented floating point precision specifications (A3)
. global default floating point precision specifiable (A3)
* ability to define new character sets (A5)
* permit non-constant array upper bounds (A6)
* assignment and access functions definable for new types (B1)

* permit functions to return array and record values (B1, E1, E2)
* programmer-definable equivalence operator for new types (B2)
. unordered enumeration types (B3)
. exponentiation operator (B4)
. define rules for truncation and rounding (B5)
. short circuit Boolean expression evaluation (B6)
* scalar operations on conformable arrays (B7)
. explicit conversion operators (B8)
* run-time exception for range errors (B9, G7, C7)
* dynamically assign/reassign I/O devices (B10)
. set complement operator (B11)
. specify left-to-right evaluation order of operands, subroutine arguments, array indices, and assignment operands (C1)
. require parentheses within the same precedence level for non-associative operators (C2)
. allow constant expressions where constants can appear (C4)
. constant expressions evaluated at compile time (C4)
* variable number of parameters (C5, C9)
* parameterized types (C5)
* parameterized array bounds (C6)
. require parameter declarations for procedures passed as parameters (C6)
* exception handling capability and control structure (C7, D4, G7)
* generic subroutines (C8)
. provide for input of Boolean constants (D2)
. require identical conversion of program and data literals (D2)
. run time testing option for accesssing uninitialized variables (D3)
. require specifying range of numeric variables (D4)
* prohibit assigning pointers to variables whose lifetime is longer than that of the object pointed to (D6)
* define new infix operators (E1)
* specification of a type's internal representation (E2)
* ability to extend existing operators to new types (E4)
* encapsulated type definition with initialization, finalization, allocation-time, and deallocation-time procedures (E5, E8)
* limit access scope (F2)
* external scope for separately defined modules (F2)
* ability to provide new local names for external identifiers (F2)
* support TINMAN library concept (F4, F5, F6)
. labels as identifiers (rather than numbers) (G2)
. restrict GOTO to same scope level and forbid jumping into loop bodies (G2)
. equivalent of Zahn's device (G3, G7)
. iteration with loop exit anywhere (G4)
* (pseudo) parallel processing capability with specifiable priorities (G6, G8)
. access to real-time clock (G8)
* synchronization, mutual exclusion, and delay primitives (G8)
. specify translation from publication language to ASCII (H3)
. internal break character for identifiers and literals (H4)
. separate quoting of each line of long literals (H4)
. uniform referent notation (H9)
. ability to specify routine is not reentrant or recursive (I2)

- provide environmental enquiries (I3)
- conditional compilation (I4)
- specify translator restrictions (I6)
* programmer control over movement between main and backing store (J1)
- machine language insertions (J3)
- associate identifiers with machine addresses (J4)
- specify development tools (K3)
- specify translator options (K4)
- special comment brackets for assertions and units of measure (K5)
- specify that syntax checking is required (L6)
- specific errors to be detected and error messages (L7)
- specify type matching rules for formal and actual parameters (M2)

Required Deletions (PASCAL)

- special parameter format for output procedures (C5)
- assignment to value parameters (C7)
- subrange data type (D4, E7)
- dynamic storage allocation procedures (F1)
- GOTO's across scope levels and not forbidding a jump into loop bodies (G2, G4)
- IF-THEN control structure (G3)
- not fully partitioned case statement (G3)
- for loop control variable accessible outside loop (G4)
- nested recursive procedures (G5)
- abbreviation of identifiers to first eight characters (H1)
- lexical units (including comments) crossing line boundaries (H5, H7)
- syntax sharing for up-arrow (H10)
- implementation dependencies (I1)

SIMULA 67 Evaluation Summary

SofTech's evaluation of SIMULA 67 with respect to the Department of Defense requirements for high order computer programming languages ("TINMAN," June 1976) is summarized here. Detailed evaluations of SIMULA 67 with respect to each TINMAN requirement are contained in separate sections of this report. These sections begin with an overall assessment of the extent to which the requirement is satisfied, followed by a justification of this assessment referencing sections of the SIMULA 67 defining document and other pertinent publications. The following documents are cited throughout the SIMULA evaluation discussions (the citation format used in these discussions is specified below for each document):

O.-J. Dahl, B. Myhrhaug, and K. Nygaard, SIMULA 67 Common Base Language, Norwegian Computing Center Publication No. S-22, 1970. ("CBL 2.2.1" refers to Section 2.2.1 in this defining document. As CBL 20 notes, the SIMULA 67 languge definition uses the revised ALGOL 60 report's syntax and semantics for features that are the same, so the following reference is also cited in this evaluation.)

P. Naur (Ed.), "Revised Report on the Algorithmic Language ALGOL 60," Comm. ACM, January 1963, vol. 6, no. 1, pp. 1-17. ("AR 3.3.5.2" refers to Section 3.3.5.2 of this document.)

G. Birtwistle, O.-J. Dahl, B.Myhrhaug, and K. Nygaard, SIMULA BEGIN, Auerbach Publishers Inc., Philadelphia, 1973. ("B 150" refers to page 150 in the SIMULA BEGIN book, which clarifies and illustrates a number of SIMULA 67 features.)

J. Palme, "SIMULA as a Tool For Extensible Program Products," ACM SIGPLAN Notices, February 1974, vol. 9, no. 2, pp. 24-40. ("Palme, p. 38" refers to page 38 of this paper)

NCC System Programming Group, "The Structure of the NCC SIMULA Compilers and Bench Mark Comparisons with other Major Languages," Norwegian Computing Center Publication No. S-43, 1972. ("Structure, p. 6" refers to page 6 of this report.)

Summary of Changes Required for SIMULA 67

We list here additions and deletions that must be made to SIMULA 67 to bring it into conformance with the TINMAN requirements. An addition is considered major (and marked with an asterisk) when it potentially interacts with many language features or when it significantly changes the character of the language and its use. Each addition or deletion is cross-referenced to a TINMAN requirement justifying the modification. A more detailed description of the nature of the required change will be found in the discussion of the referenced requirement.

Required Additions (SIMULA 67)

- problem-oriented floating point precision specification (A3)
- global default floating point precsions specifiable (A3)
- enumspecifiable character sets (A5)
- arrays indexable by enumeration types (A6)
* programmer control over how discrimination among record variants is tested (A7)
* assignment and access functions definable for new types (B1)
* return arrays and records directly as values of functions (B1, E1, E2)
* can extend domain of existing operator symbols (including equivalence) and can define new infix operators (B2, E1, E2, E4)
- same equivalence symbol for Boolean equivalence and for other forms of equivalence (B2)
- unordered enumeration types (B3)
- explicit truncation and rounding requirements defined (B5)
- exclusive-or Boolean operator (B6)
- assignment and scalar operations on arrays (B7)
- explicit conversion operators for arithmetic types (B8)
- run-time exception condition for numeric range errors (B9)
- power set of enumeration types with set operations (B11, E6)
- operand, subscript, and subroutine argument evaluation order specified as left-to-right (C1)

- require parentheses within same precedence level for non-associative operators (C2)
- evaluate constant expressions before run time (C4)
- extend class parameter capabilities to match procedure parameters (C5)
- type checking on arguments to subroutines passed as parameters (C6, C7)
* exception handling control structure plus a parameter class for specifying the control action when exception conditions occur (C7, D4, G7)
* generic procedures of more than one argument (C8)
- variable number of parameters (C9)
- constant value identifiers (D1)
- permit Boolean and NONE constants on input and output (D2)
- require identical conversion of program and data literals (D2)
- optional variable initialization instead of automatic (D3)
- run time test for uninitialized variables (D3)
- range declarations (D4)
- enumeration types (D5, E6)
* direct support for arrays of records (D5, E6)
* ability to define new infix operators (E1)
* extend existing operators to new types (E4)
* provide means of limiting the automatic inheritance of operations applicable to a data type in terms of which a new type is being defined (E5, F2)
* user-defined finalization, initialization, and deallocation actions associated with type definitions (E8)
- limit access scope both at definition and at use (F2)
- require support for application libraries (F4)
- support routines written in other source languages (F5)
- support Compool-like data sharing (F5, F6)
- more than one compool (F6)
- general case statement (G3)
- loops with exit tests in the middle and at the end (G3, G4)
- equivalent of Zahn's device (G3, G7)
* true parallel processing capability -- can create and terminate processes and have critical section control primitives (G6, G8)
* asynchronous interrupt handling (G8)
- access to real time clocks (G8)
- specify translation into ASCII character set (H3)
- break characters for internal use in identifiers and numeric literals (H4)
- separate quoting of each line of a long TEXT literal (H4)
- permit system defined procedures as parameters (H10)
- optional control over specifying object representations (I2)
- specification that a routine is not recursive or reentrant (I2)
- conditional compilations and compile time variables (I3, I4)
- make translator restrictions a part of the language definitions (I6)
* programmer control of movement between main and backing store (J1)
- machine language insertions (J3)
- ability to associate identifiers with object machine addresses (J4)
- specify object representation for composite data structures (E2, J4)
- can specify open or closed implementation for routine calls (J5)
- require support for separate compilation (K2)

- software development tools (K3)
- standard translator options (K4)
- provide for special brackets to enclose assertions (K5)
- smaller compilers for the language (L5)

Required Deletions (SIMULA 67)

- default REAL for arrays declared with no type (A1, E3)
- TEXT data type as built-in type for strings (B2, I5)
- array lower bounds being an arithmetic expression (instead of a constant) (A6)
- integer to real and real to integer conversion in assignment (B8)
- label parameters (C5, C7, G2)
- by-value parameters should be "read-only" instead of "read-write" (C7)
- by-name parameter transmission mode (C7)
- type specifications for formal parameters as a means of realizing generic procedures (C8)
- implementation restrictions on the use of the SIMSET and SIMULATION classes (E2)
- heap storage allocation (F1)
- GOTOs across scope levels (G2)
- switches (G2)
- designational expressions (G2)
- IF-THEN statements (with no required matching ELSE) (G3)
- non-fully partitioned INSPECT statement (G3)
- allowing GOTOs into loop bodies (G4)
- access to loop control variable from outside loop (G4)
- being able to define procedures within recursive procedure bodies (G5)
- allowing lexical units to cross line boundaries (H5)
- comment conventions as presently defined (H7)
- brackets (instead of parentheses) for array references (H9)
- substring assignment (H10)
- references to "operating systems" in the CBL definition (K1)
- implementation dependencies (I1)
- features that make the language non-minimal, e.g., the TEXT type for strings (I5)
- requiring garbage collection (J2)
- permitting superset implementations (L1)

TACPOL Evaluation Summary

SofTech's evaluation of TACPOL with respect to the Department of Defense requirements for high order computer programming languages ("TINMAN," June 1976) is summarized here. Detailed evaluations of TACPOL with respect to each TINMAN requirement are contained in separate sections of this report. These sections begin with an overall assessment of the extent to which the requirement is satisfied, followed by a justification of this assessment referencing sections of the TACPOL defining document:

TACPOL Reference Manual, USACSCS-TF-4-1, 15 January 1972.

Summary of Changes Required for TACPOL

We list here the additions and deletions that must be made to TACPOL to bring it into conformance with the TINMAN requirements. An addition is considered major (and marked with an asterisk) when it potentially interacts with many language features or when it significantly changes the character of the language and its use. Each addition or deletion is cross-referenced to a TINMAN requirement justifying the modification. A more detailed description of the nature of the required change will be found in the discussion of the referenced requirement.

Required Additions (TACPOL)

* * a discriminated union record capability with programmer defined discrimination tests (A1, E7)
- floating point real numbers with specifiable precision (A2, A3)
- global default floating point precision specification (A3)
- specifiable step size for fixed point numbers (A4, D4)
- modified scaling rules for fixed point arithmetic (A4)
* ability to define new character sets (A5)
- array upper bounds fixed on entry to allocation scope, instead of at compile time (A6)
- arrays indexable by enumeration types (A6)
- programmer specifiable array lower bounds (A6)
* assignment and access functions definable for new types (B1)
- return arrays and records as values of functions (B1, E1, E2)
- exact fixed point equivalence operator (B2)
- programmer-specified equivalence operator for comparing user defined types (B2)
- unordered enumeration types (B3)
- integer division with its own symbolic operator (B4)
- rounding during expression evaluation (B5, I1)
- short circuit evaluation of Boolean expressions (B6)
- scalar operations on conformable arrays (B7)
- assignments applicable to arrays and structures (B7)
- capability to dynamically assign and reassign I/O devices (B10)
- power set data type and operations (B11)
- operand, subscript, and subroutine parameter evaluation done left-to-right (C1)
- parentheses required at same precedence level for non-associative operators (C2)
- explicit statement of where expressions can occur (C3)
- allow constant expressions wherever constants are allowed (C4)
- evaluate constant expressions before run time (C4)
- permit procedures passed as parameters to take arguments (C5)
* variable number of procedure parameters (C5, C9)
* exception handling parameter class (C7)
* a generic procedure capability (C8)
- require identical conversion of program and data literals (D2)
- permit records as comonents of records (D5, E6)
- initialization facility (D3)
- generalized range declarations (D4)
- enumeration types (D5, E6)

. pointer variables (D6)
* user definition of new types and infix operators (E1)
. consistent use of types (E2)
. require declaration of loop control variable (E3)
* user can extend existing operators (E4)
* data types with clustered operations definable (E5)
* type initialization and finalization (E8)
. provide for separating variable allocation from variable access (F1)
* ability to limit access to separately defined structures (F2)
. associate new local names with separately defined components (F2)
. application oriented libraries (F4, F5)
. routines written in languages other than TACPOL (F5)
. Compools at all levels of programming activity (F6)
. more than one Compool (F6)
. case statement (G3)
. loops with exit tests in middle (G3, G4)
. equivalent of Zahn's device (G3, G7)
. recursive procedures (G5)
* parallel processing capability (G6)
* fuller exception handling (G7, B9, D4)
* asynchronous interrupt handling (G8)
. access to real-time clocks (G8)
* synchronization and real-time capability (G8)
. break character for use internal to identifiers and literals (H4)
. separate quoting of each line of a long literal (H4)
. require key words to be reserved (H6)
. compile time variables (I3)
. conditional compilation capability with encapsulated code (I4, J3)
. ability to associate identifiers with object machine addresses (J4)
. greater control over composite object representation (J4)
. specify procedure call as open or closed implementation (J5, I2)
. more extensive translator options and available tools (K3, K4)
. special comment brackets for assertions (K5)
. suggested set of diagnostic messages (L7)
. language documentation, especially a formal in-depth definition (M3)

Required Deletions (TACPOL)

. character and bit string types (A2, B11, I5)
. MOVE operation to be replaced by a type checking structure assignment (A6, B1, B7)
. overlay with free union capability (B8, E7)
. partitioned data sets and file labels (B10)
. label parameters (C7, G2)
. write access to value parameters (C7)
. requiring specification of parameter type and dimension (C8)
. restrictions on nesting arrays and structures (D5)
. GOTOs out of same scope (G2)
. switches (G2)
. if-thens (with no else) (G3)
. DO control variable being non-local to DO loop (G4)
. GOTO into loops (G4)
. abbreviation of identifiers (H1, I1)

- continuation of lexical units across line boundaries (H5)
- size of key word list (H6)
- comments crossing line boundaries (H7)
- SUBSTR assignable pseudo-variable (H10)
- syntax sharing of equal sign for both equality test and assignment (H10)
- implementation dependencies (I1)
- multiple target assignments (I5)

A1. Typed Language (TINMAN)

The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs.

By the type of an object is meant the set of objects themselves, the essential properties of those objects, and the set of operations which give access to and take advantage of those properties. Compile time determination of types does not preclude the inclusion of language structures for dynamic discrimination among alternative record formats (see A7) or among components of a union type (see E6). Where the subtype or record structure cannot be determined until run time, it should still be fully discriminated in the program text so that all the type checks can be completed at compile time.

A1. Typed Language (ALGOL 68)

Satisfied.

Algol 68 is a typed language [Section 2.1.1.2, 1.2.1, 4.4, 4.6]. All identifiers must be declared [Section 4.4] with their types (modes). Labels are implicitly declared by their placement (prefixing a statement) [Section 3.2]. Field selectors are part of the mode of composite structures [Section 1.2.1]. The type of each language element is known at compile time, except for union types, which are fully discriminated in the program text, using case conformity clauses [Section 3.4.1q].

A1. Typed Language (J3B)

Partially Satisfied.

Explicit declarations for variables and components of composite data structures are always required, even for loop control variables [Sections 3.2 and 4]. Declarations of labels are required when necessary to override a macro declaration declared in a higher scope [Section 5.4.2]. However, there is no language construct that enforces and controls run-time discrimination among the subtypes of a record structure in the program text, so access to

records with alternative structures cannot be validated fully at compile-time. Even so, the requirement that fields of records be accessed only through pointers of an appropriate (possibly union) type [Section 6] means that compile-time checks on access to record alternatives are required to some extent.

A1. Typed Language (PASCAL)

Partially Satisfied.

"Every variable occuring in a statement must be introduced by a variable declaration" [p. 137]. Each component of a record has its type specified in the program text [p. 144]. Even labels must be explicitly declared [p. 158-9]. However, PASCAL provides no language construct for discriminating among record variants whose discrimination condition is not determined by a field in the record (see E7).

A1. Typed Language (SIMULA 67)

Partially Satisfied.

Explicit type declarations of variables are required except in one case: in an array declaration "if the type is omitted, the array is assumed REAL by default" [B 373]. "The structure of the SIMULA language is such that almost all the error checking of a program can be done at compile time" [Palme, p. 38]. Type checks can be performed at compile time in the sense required by A1, i.e., in the case where mismatching between the type required to perform an operation correctly, and the type actually needed, type checks can either be performed at compile time or must be specified in the program text via the INSPECT statement [B 150].

A1. Typed Language (TACPOL)

Partially Satisfied.

"All names must be declared" [Section 10-1]. Moreover, "the type of a quantity and the type of a value are explicitly defined" [Section 2-6.c]. However, the TACPOL CELL capability [Section 4-7.a] permits alternative record structures to be overlayed without requiring or providing constructs permitting the programmer to fully discriminate among these alternatives in the program text.

A2. Data Types (TINMAN)

The language will provide data types for integer, real (floating point and fixed point), Boolean, and character, and as type generators, will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e.,

composite data structures with labeled components of heterogeneous type).

A2. Data Type (ALGOL 68)

Partially Satisfied.

Algol 68 has integer, floating point real, Boolean, and character types, along with arrays and structures [Section 2.1.1.2, 1.2.1, 4.6]. It does not have a fixed point real type. Real constants [Section 8.1.2] and I/O formats [Section 10.3.4.3] are available. Algol 68 also has the following types: pointer (to another type), procedure (with typed parameters and results), (variable length character) string (flexible size array of character), bits (fixed size array of Boolean), and bytes (fixed size array of character).

A2. Data Types (J3B)

Partially Satisfied.

J3B provides integer, floating point (single and double precision), fixed point, character, array and record data types [Sections 4.2-4.5 and 4.8]. Only type generators for records are supported [Section 4.2]. J3B does not directly support a Boolean data type. Instead, bitstrings are given a Boolean interpretation in appropriate contexts [Sections 5.5 and 5.6]. Fixed length character strings are supported directly as a built in data type, although I5 implies character strings are to be supported as arrays of characters.

A2. Data Types (PASCAL)

Partially Satisfied.

PASCAL provides integer, real, Boolean, and character as standard types [p. 142]. Arrays and records are available as structured types [p. 143]. However, floating point is the only real representation supported [p. 93]. PASCAL does not have a fixed point real data type. PASCAL supports (fixed length) character strings as packed arrays of characters [p. 141].

A2. Data Types (SIMULA 67)

Partially Satisfied.

SIMULA 67 has REAL, INTEGER, BOOLEAN [AR 5.1.1], CHARACTER [CBL 3.1], pointer or REFerences to objects of a CLASS, which effectively realizes records [CBL 3.1], and ARRAY types [CBL 3.1w]. But fixed point is not supported [B 71].

The SIMULA text data type provides fixed length character strings (with programmer-defined upper limit on length defined at run time) and several associated operators [CBL 10, B 178].

A2. Data Types (TACPOL)

Partially Satisfied.

TACPOL has fixed point reals, integers (i.e, fixed reals with scale of zero), character strings, and bit strings as basic data types [Section 3-2, 3-3, 3-4, 2-6.h.(2)]. Arrays [Section 4-5, 4-6] and a form of records [Section 4-6, 4-7] are available, but not as type generators. There are restrictions on arrays and records that limit their generality (see D5, A7). TACPOL's character and bit string types exceed A2's requirement for Boolean and character (non-string) types. TACPOL does not support a floating point real type.

A3. Precision (TINMAN)

> The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

Precision specifications will not change the type of reals nor the set of applicable operations. Precison specifications apply to arithmetic operations as well as to the data and therefore should be specified once for a designated scope. This permits different precisions to be used in different parts of a program.

A3. Precision (ALGOL 68)

Not Satisfied.

Algol 68 has variable precision in that integral and real types may be declared as real, long real, long long real, short real, etc. indicating more or less precision than real. However the number of bits (digits) corresponding to real, long real etc. is implementation dependent.

There is no way to specify the precision of arithmetic operations in some scope.

A3. Floating Point Precision (J3B)

Not Satisfied.

Although J3B supports single and double precision arithmetic [Sections 1.3.3.1], it does not support any more specific form of precision specification. Nor does it provide a means of specifying global (to a scope) precision for floating point arithmetic operations.

A3. Precision (PASCAL)

Not Satisfied.

PASCAL has only one representation for real variables [p. 93, 142]. There is no way for the programmer to specify precision, not even the common single or double precision.

A3. Precision (SIMULA 67)

Not Satisfied.

SIMULA 67 has an ALGOL 60 form of reals [B 71, AR 5.1.3]. Not even FORTRAN-like double precision can be specified. Accuracy is not explicitly definable. In fact, it is classified as "implementation defined" [B 72]. Global specification of precision is not supported.

A3. Precision (TACPOL)

Not Satisfied.

TACPOL does not support a floating point real data type.

A4. Fixed Point Numbers (TINMAN)

Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.

Integers will also be treated as exact quantities, with a step size equal to one.

A4. Fixed Point Numbers (ALGOL 68)

Not Satisfied.

Algol 68 does not have a fixed point real type [Section 1.2.1, 4.6].

A4. Fixed Point Numbers (J3B)

Not Satisfied.

Although J3B has a fixed point data type, it is a "fractional" rather than exact representation [Section 1.3.3.1, p. 1-14]; the declaration format does not permit an implementation independent specification of a step size, but instead requires the programmer to specify the assumed position of the binary point with respect to a computer word [Section 4.8]. For example, the

fixed point specification A 4 implies the binary point is 4 binary digits to the right of the sign bit. All fixed point values occupy the same number of bits, i.e., they all have the same precision.

Scaling of fixed point computations is performed automatically by the compiler [Section 7.1.4], with a programmer option for performing the scaling explicitly [Section 9.3]. The default scaling rules are not defined such that operations on fixed point values types having (in effect) a step size of one yield the same result as operating on integer value types. Moreover, the default scaling rules do not preserve fractional bits in arithmetic and comparison operations, demonstrating J3B's support of a "fractional" rather than an "exact" concept of fixed point values.

A4. Fixed Point Numbers (PASCAL)

Not Satisfied.

PASCAL does not support computation with fixed point real numbers [p. 93].

A4. Fixed Point Numbers (SIMULA 67)

Not Satisfied.

Fixed point reals are not supported [B 71].

A4. Fixed Point Numbers (TACPOL)

Partially Satisfied.

"In TACPOL, fixed point decimal numbers are converted and manipulated as their binary fixed point equivalents" [Section 3-2], meaning that only integer multiples of powers of two are represented exactly. (In terms of the TINMAN requirement, only step sizes that are powers of two are supported.) Numeric literals that are not integer multiples of powers of two are permitted, however, and are implicitly converted to an approximation of the decimal value. In this sense, TACPOL does not support an "exact" fixed point data type, since a programmer is not constrained to use literals exactly representable as integer multiples of a particular step size.

"During the evaluation of expressions, the scale factors of the values are automatically adjusted to the scale factor of the value containing the largest fractional part" [Section 6-2]. Presumably this implies that the product of two fixed point values with a scale of, say, 3 is also 3, so three fractional bits are lost. Hence, the scaling rules for multiplication and division do not conform to those of an "exact" fixed point data type.

Only fixed point precisions of 8, 15, 31, and 62 are supported [Section 3-2] by the implementation (although precisions of 1 to 62 can be specified in

declarations). Negative values cannot be represented if values are represented with a precision of 8. The precision of the result of arithmetic operations is either 31 or 62 bits. The precision is 62 if one of the operands has a specified precision of 32 to 62, or if both operands have a specified precision of 1 to 31 and the long precision multiply or long precision exponentiation operator is used [Section 6-2].

The scaling rules for comparison of fixed point values with different scales [Section 6-1.1] insure that no fractional bits are truncated before the comparison is performed (which is consistent with "exact" fixed point concepts), but no check is made on the possible loss of significant bits, so incorrect results can occur [Section 6-1.b(1)].

A5. Character Data (TINMAN)

Character sets will be treated as any other enumeration type.

Like any other data type defined by enumeration (see E6), it should be possible to specify the program literal and order of characters. These properties of the character set would be unalterable at run time. In general, unless all devices use the same character code, run-time translation between character sets will be required. Widely used character sets, such as USASCII and EBCDIC will be available in a standard library.

A5. Character Data (ALGOL 68)

Not Satisfied.

The collating sequence (mapping from char to int) is implementation defined [Section 2.1.3.1.g] in Algol 68. The language does not have enumeration types. However, one can construct other collating sequences and translate between them since the language allows arrays of characters and has the operators abs and reps [Section 10.2.1.n-o] for converting between char and int types.

A5. Character Data (J3B)

Not Satisfied.

Character strings are supported by the language, but in an implementation defined code [Section 1.3.3.1, p. 1-15]. The programmer has no ability to define new character sets, nor does J3B support an enumeration data type.

A5. Character Data (PASCAL)

Not Satisfied.

The programmer can define a scalar type by enumeration of identifiers [p. 142]. This enumeration defines an ordering to which both a successor and a predecessor function, as well as standard relational operators [p. 151], are applicable. Moreover, a variable of such a programmer enumerated scalar type can be used as an array index [pp. 143, 146]. For the standard scalar type "char" the ordering is implementation determined [pp. 142, 94-95]. However, a non-standard character set cannot be directly treated as a scalar type since only identifiers can be enumerated [p. 142].

The CDC ASCII 64 character set (which differs from the ISO standard) is available [p. 94] as a standard character set. And built-in functions chr (n), giving the n-th character in the standard scalar type char, and its inverse, ord, may be used in programmer defined conversions [p. 164].

A5. Character Data (SIMULA 67)

Not Satisfied.

SIMULA 67 does not support "enumeration type" as in, say, PASCAL. "The set of internal characters is ordered according to an implementation defined collating sequence" [CBL 3.2.2.1]. The built-in function procedure rank and char provide one-one conversion between integer character codes, such as EBCDIC, and characters [CBL 3.2.2.1, B 175-176]. This can facilitate translation between character sets. But the programmer cannot specify the "order of characters" (TINMAN A5).

A5. Character Data (TACPOL)

Not Satisfied.

TACPOL does not support defining new types by enumeration. The collating order of characters is predetermined and used for relational operators [Section 6-1.b(2)]. The programmer cannot specify or change this order.

A6. Arrays (TINMAN)

> The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type, and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

The range of subscript values for any given dimension will be a contiguous subsequence of values from an enumeration type (including integers). The preferable lower bound on the subscript range will be the initial element of an enumeration type or zero, because it often contributes to program efficiency and clarity.

A6. Arrays (ALGOL 68)

Partially Satisfied.

In Algol 68 the number of dimensions and the element type are part of the array type [Section 4.6, 2.1.3.4] and are determinable at compile time. The lower and upper bounds may be expressions and hence their values need not be defined until run time. For non-flexible arrays, the bounds remain fixed once the array is allocated. For a variable declared as a flexible array, the bounds may be varied by assigning arrays with differing numbers of elements to the variable. Subscripts must be of type integer; there are no enumeration types. The default lower bound is 1.

A6. Arrays (J3B)

Partially Satisfied.

All requirements are satisfied except that the upper subscript bound is determined at compile time, not on entry to the array allocation scope, and the lower subscript bound is not programmer definable (it is fixed at zero) [Section 4.4]. Note, however, that lower bounds of tables (i.e., arrays of records) can be specified to be a constant other than zero [Section 4.3].

A6. Arrays (PASCAL)

Partially Satisfied.

PASCAL does not meet the crucial requirement that "the upper subscript bound will be determinable at entry to the array allocation scope" since the bounds (both upper and lower) of an array dimension are fixed at compile time [p. 143] even though arrays may be allocated at run time on entry to a procedure [p. 159]. PASCAL is compatible with the other A6 array requirements. The default lower bound for arrays is one.

A6. Arrays (SIMULA 67)

Partially Satisfied.

SIMULA 67 satisfies A6 except that the lower subscript bound is not determinable at compile time. Rather, "each bound is an arithmetic expression" [B 371, AR 5.2.1], for example, the variable N. Further, with no enumeration type facility, array subscripts are restricted to integer values [AR 3.1.4.2].

A6. Arrays (TACPOL)

Partially Satisfied.

In TACPOL, "three dimensions are the maximum allowable within an array declaration" [Section 4-2.c]. The range of subscript values is fixed at compile time. Lower subscript bounds are not programmer-definable, and are assumed to be 1 [Section 4-2.a].

A7. Records (TINMAN)

The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

This permits hierarchically structured data of heterogeneous type, permits records to have alternative structures as long as each structure is fixed at compile time and the choice is fully discriminated at run time, but it does not permit arbitrary references to memory nor the dropping of type checking when handling overlayed structures. The discrimination condition will not be restricted to a field of the record but should be any expression.

A7. Records (ALGOL 68)

Partially Satisfied.

Component names (field selectors) are part of the structure type, as are the types of the component fields [Section 1.2.1, 4.4, 4.6, 2.1.3.3], and hence must be declared. Alternate structures are obtained by declaring a field to be a union type. Algol 68 union types are fully discriminated, but the programmer has no control over how the discimination condition is represented.

A7. Records (J3B)

Partially Satisfied.

Records with alternative structures are defined in J3B as "typed tables" [Section 4.3]. However, testing to discriminate at run time among record alternatives is not directly supported by a language construct.

A7. Records (PASCAL)

Partially Satisfied.

A PASCAL record type can have several variants with each identified and typed in its declaration [p. 144]. However, the requirement that the "discrimination condition will not be restricted to a field of the record but should be any expression" (TINMAN, p. 21) implies a discriminated union type union facility without necessarily having a discrimination tag as a record field (or even stored within the record representation). In PASCAL the tagfield of a variant record is optional [p. 144]. When it is omitted, the equivalent of a free type union is obtained [Wirth, p. 197].

A7. Records (SIMULA 67)

Partially Satisfied.

SIMULA 67 provides the equivalent of records with alternative structures with its prefixed classes (or subclasses) capability [CBL 2.2.1]. The INSPECT statement with its optional WHEN form [CBL 7.2.1, B 150-151] provides for discrimination between objects (i.e., record equivalents) of alternative structures. The programmer does not, however, have control over the internal representation of the discrimination information.

A7. Records (TACPOL)

Not Satisfied.

TACPOL severely limits the hierarchical structuring of heterogeneous data in that records and arrays of records may not be components of records. Record components are limited to scalars and arrays of elementary types [Section 4-5]. A special construct, CELL, permits records to be overlayed [Section 4-7.a]. This provides a minimal variant record capability. Discrimination and type checking are the programmer's responsibility.

B1. Assignment and Reference (TINMAN)

Assignment and reference operations will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

The user will be able to declare variables for all data types. Variables are useful only when there are corresponding access and assignment operations. The user will be permitted to define assignment and access operations as part of encapsulated type definitions (see E5). Otherwise, they will be automatically defined for types which do not manage the storage for their data. (See D6 for further discussion).

B1. Assignment and Reference (ALGOL 68)

Partially Satisfied.

The Algol 68 user may declare variables of any type [Section 4.4]. Assignment [Section 5.2.1] and access [Section 4.8, 6.2] are automatically available for any type. However, assignment and access are not operators and cannot be user defined. But the user may define other operators to have the same effect as assignment and access for specific types.

Arrays and records can be specified as values of functions [Section 5.4.1] although the implication of such a specification is that a pointer to an array or record is what is actually returned.

B1. Assignment and Reference (J3B)

Partially Satisfied.

Although assignment and reference operations are automatically defined for all J3B data types, and assignments are permitted among records with alternative structures (which can be used to, in effect, implement a union data type), the user is not permitted to define assignment and access operations as part of record type definitions.

In addition, arrays and records cannot be returned as values of functions [Section 3.3.3], although such a capability is implied by B1, E1, and E2.

B1. Assignment and Reference (PASCAL)

Partially Satisfied.

In PASCAL "direct assignment is possible to variables of any type, except files" [p. 21]. Files are a type in PASCAL [p. 145] but presumably files are not considered to manage their data storage".

Arrays and records cannot be specified as values of functions (p. 162), although such a capability is implied by B1, E1, E2, and B7.

No ability to define assignment and access operation for new types is provided.

B1. Assignment and Reference (SIMULA 67)

Partially Satisfied.

All elementary and character string (i.e., text) values can be assigned. Other data types are essentially objects of declared classes, with object reference assignment automatically defined to cover any user defined classes [CBL 6.1, 6.1.2, 6.1.2.2]. However, the user cannot define assignment and access operations that are automatically invoked by, say, the assignment operator.

Arrays and records cannot be returned as values of functions [CBL 8.1], although such a capability is implied by B1, E1, E2, and B7.

B1. Assignment and Reference (TACPOL)

Partially Satisfied.

The assignment operator applies only to elementary types (i.e, short and long fixed point numerics, character strings, and bit strings) [Section 5-1]. A built-in procedure MOVE can be employed to assign structured types (i.e, arrays, groups, tables, and cells). MOVE effects bit string copying and does no type checking [Section A-6]. Of course, assignment of file types is not supported [Section 13-16]. Encapsulated type definitions are not present in TACPOL (see E5).

Arrays and records cannot be returned as values of functions, although B1, E1, E2, and B7 together imply the need for such a capability.

B2. Equivalence (TINMAN)

The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

There are many useful equivalence operations for some types and a language definition cannot foresee all these for user defined types. Proper semantic interpretation of identity requires that equality and identity be the same for atomic data (i.e., numbers, characters, Boolean values, and types defined by enumeration) and that elements of disjoint types never be identical. For floating point numbers identity will be defined as the same within the specified (minimum) precision.

B2. Equivalence (ALGOL 68)

Satisfied.

The Algol 68 equal operator (=) is defined for integer, real, Boolean, and character types [Section 10.2.3]. The identity relation (:=:) [Section 5.2.2] allows one to test whether two names (addresses) are the same. The programmer may define the = operator for other types, e.g., arrays, records (structures) and mixed precision real or integer.

B2. Equivalence (J3B)

Partially Satisfied.

An equivalence operation is automatically defined for comparing integer, floating, fixed point, bit string, character string, and pointer variables and literals [Section 7.3]. In addition, table entries (i.e., elements of an array of records) can be compared for equality. However, a user cannot define the meaning of equivalence for a record data type (the only user-definable data type permitted in J3B), and floating point values of different precisions are not compared within their minimum precision (e.g., in comparing a single

precision quantity with a double precision quantity, the double precision value is not truncated before comparison; instead the single precision value is compared against the exact double precision value) [Section 7.3, p. 7-14].

B2. Equivalence (PASCAL)

Partially Satisfied.

The built-in equivalence relational operator (=) is only applicable to operands of scalar, subrange, or [packed] array of character (i.e., strings) type [p. 151]. The equivalence operator doesn't encompass the other structured types (e.g., records) and a programmer cannot extend its definition to these types. Since only one floating point precision is supported, the question of how to compare values of differing precisions does not arise.

B2. Equivalence (SIMULA 67)

Partially Satisfied.

EQV is the logical equivalence operator [B 369] and SIMULA 67 inherits the ALGOL 60 equivalence relation for simple arithmetic expressions [AR 3.4.1] including reals. This implies "bit-by-bit comparison" for reals [AR 3.3.6]; this satisfies B2 since only one real precision is supported. Character and text values are compared for being equal (not identical) [CBL 5.1, 5.2]. References are compared for identity (i.e. if they refer to the same object or text value instance) with == and =/= [CBL 5.4]. In this vein if we let T and U be text references, then "relations T=/=U and T=U may both have the value true. Then T and U refer to physically distinct character sequences which are equal" [CBL 5.4.2]. The distinction between EQV and = means SIMULA does not have a single built in equivalence operator for atomic data.

The definition of built in infix operators (in particular, =) cannot be extended to new data types.

B2. Equivalence (TACPOL)

Not Satisfied.

The built-in equivalence operator tests only for equality of elementary (numeric or string) types [Section 6-1.b]. In the comparison of arithmetic values "the possibility of some significant bits being lost" through shifting can produce an unintended true result [Section 6-1.b.(1)]. Comparing structured types must be coded since it is not built-in.

B3. Relationals. (TINMAN)

Relational operations will be automatically defined for numeric data and all types defined by enumeration.

It will be possible to inhibit ordering definitions when uxnrdered sets are intended.

B3. Relationals (ALGOL 68)

Partially Satisfied.

All six relations are defined for integer, real, character, and Boolean types [Section 10.2.3]. They are also defined for the string (flexible size character arrays) and bytes (fixed size character arrays) types. Algol 68 has neither enumeration types nor unordered set types.

B3. Relationals (J3B)

Partially Satisfied.

Relational operations are defined for numeric data in J3B [Section 7.3]. Only equality and inequality comparisons, however, are permitted among character strings [Section 7.3], and since J3B does not support an enumeration data type, there are no relational operations applicable to such data. In addition, J3B permits relational comparisons between pointer variables of compatible type [Section 7.3]. (This is because in J3B, pointers can be used to access table entries, i.e., elements of an array of records, and therefore, operations on index values are extended to pointer values.)

B3. Relationals (PASCAL)

Partially Satisfied.

The six relational operators are applicable to operands of any scalar or subrange type [p. 151]. In PASCAL scalar types include not only number and enumeration but also character and Boolean [p. 151] which, in effect, inhibit any ordering of set elements.

PASCAL does not support unordered enumeration types, however.

B3. Relationals (SIMULA 67)

Partially Satisfied.

All six relational operators are defined for numerics and text data [AR 3.4.1, CBL 5., B 183]. SIMULA 67 does not have types defined by enumeration and, hence, B3 is vacuously satisfied for such types.

B3. Relationals (TACPOL)

Partially Satisfied.

All six relational operators are included in TACPOL and are applicable to numeric and string types [Section 6-1.b]. Defining types by enumeration is not supported.

B4. Arithmetic Operations. (TINMAN)

The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

Floating point operations will be precise to at least the specified precision.

B4. Arithmetic Operations (ALGOL 68)

Partially Satisfied.

Algol 68 has all of the required operations [Section 10.2.3]. However, it has no fixed point type. The exponentiation operator is defined only for real bases and integer exponents, but may be user defined for other combinations of numeric types. Other arithmetic dyadic operators are not defined over operands of differing precision, e.g., there is no built-in definition for adding a real and a long real [Section 10.2.3.4, 10.2.3.5].

B4. Arithmetic Operations (J3B)

Partially Satisfied.

Addition, subtraction, multiplication, division, and negation operators are provided for numeric data [Section 7.1]. Exponentiation is supported, but only yields a floating point result [Sections 7.1.2 and 7.1.3]. In addition, exponentiation cannot be applied to fixed point values [Section 7.1.4]. Implementations that do not support floating point data also do not support an exponentiation operator [Section 7.1].

There is no operation that directly yields the remainder when two integer values are divided [Section 7.1.1].

The result of floating point computations are maintained to a precision equal to the greatest precision of the operands [Sections 7.1.2 and 7.1.3].

B4. Arithmetic Operations (PASCAL)

Partially Satisfied.

PASCAL has all of the required built-in arithmetic operations except exponentiation [p. 150].

B4. Arithmetic Operations (SIMULA 67)

Satisfied.

All of the required built-in arithmetic operators are inherited from ALGOL 60 [CBL 4.1.1, AR 3.3.1]. The programmer cannot specify the precision for floating point operations.

B4. Arithmetic Operations (TACPOL)

Partially Satisfied.

TACPOL has the required built-in functions except for integer division [Section 6-1.a]. But only "exponentiation by positive integers is permitted" [Section 6-1.a]. Integer division can be effected by assigning a fixed real division result to an integer (i.e, scale factor of zero). A division remainder built-in function REM is provided for both short numeric [Section A-2] and long numeric [Section A-3] values.

B5. Truncation and Rounding (TINMAN)

Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

B5. Truncation and Rounding (ALGOL 68)

Partially Satisfied.

Assignment [Section 5.2.1] and access [Section 4.8] never truncate or round in Algol 68. However, truncation and rounding are implementation dependent for the arithmetic operations [Section 10.2.3, 2.1.3.1]. The round operator is only defined for converting a real value to an integer value of equivalent precision [Section 10.2.3.4], while the shorten operator applies to reals, yielding a rounded real result of less precision. The entier operator truncates reals to integer values. There is a widening coercion [Section 6.5] that implicitly converts integers to reals.

B5. Truncation and Rounding (J3B)

Partially Satisfied.

Arithmetic and assignment operations on integer, floating and fixed point quantities do not truncate significant digits as long as overflow does not occur [Section 7.1 and 5.2]. Implicit truncation of low-order digits is permitted, however, in fixed and floating point arithmetic and assignments [Section 7.1 and 5.2]. Rounding is not required and there is no explicit ROUND operator. In assigning fixed or floating point values to integer variables, fractional bits are always truncated implicitly [Section 5.2, p. 5-5].

B5. Truncation and Rounding (PASCAL)

Not Determinable.

It is not specified as part of standard PASCAL how truncation and rounding are done during arithmetic and assignment operations. However, PASCAL does contain standard built-in transfer functions trunc () and round () for converting from real to integer [p. 164].

B5. Truncation and Rounding (SIMULA 67)

Partially Satisfied.

Reals are implicitly rounded [AR 4.2.4]. But "it is understood that different hardware representations may evaluate arithmetic expressions differently" [AR 3.3.6]. SIMULA 67 does not have fixed point reals, nor is precision definable for floating point reals.

B5. Truncation and Rounding (TACPOL)

Not Satisfied.

"During the evaluation of expressions, the scale factors of the values are automatically adjusted to the scale factor of the value containing the largest fractional part" [Section 6-2.a]. This means the most significant bits can be implicitly truncated and that low order bits can be lost on multiplication. TACPOL does provide built-in TRUNC and ROUND procedures for short [Section A-2] and long [Section A-3] numeric values.

B6. Boolean Operations (TINMAN)

The built-in Boolean operations will include "and, " "or, " "not, " and "xor." The operations "and" and "or" on scalars will be evaluated in short circuit mode.

Note that the equivalence and nonequivalence operations (see B2) are the same as logical equivalence and exclusive-or respectively.

B6. Boolean Operations (ALGOL 68)

Partially Satisfied.

Algol 68 has "and", "or", "not", and "exclusive or" operators [Section 10.2.3.2]. However, the order of evaluation of operands is not defined [Section 5.4.2]. Choice clauses (i.e. conditional expressions) must be used to obtain the effect of short circuit evaluation in an implementation independent manner, e.g., short circuit A AND B could be written (A|B|false).

B6. Boolean Operations (J3B)

Partially Satisfied.

J3B does not support Boolean data types, but instead, supports bit strings. AND, OR, NOT, and XOR are available as operations on bit strings [Section 7.3]. Short circuit evaluation of bit string operations is permitted but not required by the language specification (p. 7-14).

B6. Boolean Operations (PASCAL)

Partially Satisfied.

The Boolean operators "and," "or," and "not" are built-in and "xor" for Booleans is a special case of the inequality operator [pp. 149-150].

However, PASCAL does not evaluate Boolean expresssions in short circuit mode. In coding "and" or "or" operators "the programmer must assure that the second factor is well-defined, independent of the value of the first factor" [p. 21].

B6. Boolean Operators (SIMULA 67)

Partially Satisfied.

Exclusive-or is not a built in Boolean operator; only EQV is built-in. The operators AND and OR are defined to be evaluated in short circuit mode (e.g., P AND Q is defined to mean IF P THEN Q ELSE FALSE) [B 369]. (Note: Boolean relations are also defined as ALGOL 60 relations [CBL 5] and the ALGOL 60 and operator definition does not require short circuit mode evaluation [AR 3.4.5].)

B6. Boolean Operations (TACPOL)

Partially Satisfied.

TACPOL has the Boolean operators AND, OR, and NOT built-in [Section 6-1.d]. They apply to bit strings and yield bit string results. The not equal NE relational operator yields a one bit result [Section 6-1.b.(3)]. But exclusive-or (and any other Boolean operation) can be realized using the built-in bit string procedure BOOL [Section A-5]. Short circuit evaluation mode is not required.

B7. Scalar Operations on Composite Data Structures (TINMAN)

The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

Conformability will require exactly the same number of components (although a scalar can be considered compatible with any array) and one for one compatibility in type. Correspondence will be by position in similarly shaped arrays. Although component by component operations will be available for built-in composite data structures which are used to define application-oriented structures, that capability will not be automatically inherited by defined data structures. Component by component operations also allow operations on character strings represented as vectors of characters and allow efficient Boolean vector operations.

B7. Scalar Operations on Composite Data Structures (ALGOL 68)

Partially Satisfied.

Algol 68 allows assignment of arrays and records (structures) [Section 5.2.1]. Each logical structure has only one physical structure since there are no packed or aligned attributes. For array assignment, the bounds must match, but the programmer can use the new lower bound option to force a match. Thus, only the subscript extent (for each dimension) must match. The scalar operators are not defined for arrays and structures, but the programmer can define them for these types. However, if one defines, for example, multiplication of arrays to be component by component, then this definition is inherited by types (e.g., matrix) represented as arrays.

B7. Scalar Operations on Composite Data Structures (J3B)

Partially Satisfied.

Although J3B permits an array element which is a record to be compared for equality with a similar array element (or the value zero) [Section 7-12], and although assignments to such elements are permitted [Section 5.2], a general form of assignment is not supported for arrays and records as a whole.

In addition, permitted assignments of array elements do not necessarily require one-for-one compatibility in type (p. 5-6). Furthermore, no scalar operations can be applied to arrays or records as a whole.

B7. Scalar Operations on Composite Data Structures (PASCAL)

Partially Satisfied.

Scalar operations cannot take arrays as operands [p. 150]. Direct assignment can be applied to arrays or records as long as the transfer is between identical types [p. 21]. This should permit run time conversion of object representations since the prefix packed for arrays or records "has no effect on the meaning of a program" [p. 143]. But, in fact, "a component of a packed structure must not appear as an actual variable parameter" [p. 83].

B7. Scalar Operations on Composite Data Structures (SIMULA 67)

Not Satisfied.

Neither scalar operations nor assignment on conformable arrays are permitted. For example, in arithmetic assignment the left side is restricted to a variable of type real or integer [CBL 6.1.2.1]. That is, an array is not permitted as the left side (receiver) of an assignment. Data transfers for aggregates are not possible. The built-in COPY function for text-objects produces only identical copies, not some conversion or packing [B 179-181].

B7. Scalar Operations on Composite Data Structures (TACPOL)

Not Satisfied.

In TACPOL, assignment [Section 5-1] and scalar operations [Section 6-1] are applicable only to elementary numeric and string data types. The built-in procedure MOVE copies the bits comprising an aggregate's value [Section A-6]. As such, it partially supports array and structure assignment, but no type checking is done.

B8. Type Conversion (TINMAN)

There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representations of numbers and their representations as characters, and between fixed point scale factors.

B8. Type Conversion (ALGOL 68)

Not Satisfied.

Algol 68 has several coercions (implicit type conversions), but since Algol 68's concept of type is somewhat broader than the TINMAN's, only widening violates B8. Widening [Section 6.5] transforms integers to reals in assignments and subroutine calls, but not in loops or array subscripting. Dereferencing [Section 6.2] obtains the value stored at a given address. (Note: the type of an Algol 68 variable is "reference to another type", so a dereferencing coercion is extremely useful.) Deproceduring [Section 6.3] allows a parameterless procedure to be called without having to write an empty argument list "()". Uniting [Section 6.4] converts a component type of a union type to the union type itself. The union type may be fully discriminated by a conformity clause [Section 3.4]. Rowing [Section 6.6] forms a one element array from a single element. Voiding [Section 6.7] discards the value of a unit (expression or statement). (In a serial clause (block body) [Section 3.2], all units (statements) except the last are voided.) Algol 68 has no fixed point type. The language has operators for converting between real and integer values of differing precision [Section 10.2.3]. It also has two real to integer conversion operators, round and entier. There are procedures for converting between numeric and character representations of numbers [Section 10.3.2.1].

Relational operations on strings of different lengths are permitted [Section 10.2.3.10]. Assignment of strings never, however, causes truncation of the string being assigned.

B8. Type Conversion (J3B)

Partially Satisfied.

The J3B parameter passing rules for typed tables and typed pointers (to tables) satisfy the TINMAN parameter matching constraints in that all the fields of a record accessible through a formal parameter must be potentially present in the record (or array of records) serving as the actual parameter [Section 5.3.3, p. 5-12].

On the other hand, since J3B also supports untyped tables and pointers, the TINMAN's parameter matching constraints can be violated. Moreover, the language supports implicit conversion from integer to real and real to integer in assignment statements [Section 5.2, p. 5-4], although such conversions are not permitted between actual and formal parameters [Section 5.3.3]. Explicit conversion operators are provided for converting between integer and fixed point representations [Section 9.4], but no explicit operators are provided for converting to or from floating point representations. There are also no conversion operators to convert between the object representation of numbers and their representations as character strings, although the BYTE and INTR operation [Section 9.2] can be used to program such conversion operations.

An operator is provided for converting between fixed point scale factors [Section 9.3]. However, the operator does not override the default fixed point scaling rules, but instead rescales the result of the default rules. This is not the most appropriate form of rescaling operator.

The intent of B8 is also not satisfied by implicit lengthening and shortening of bit and character strings in assignment operations and relational comparisons [Section 5.2, p. 5-5].

The requirement is satisfied with respect to array indexing, since implicit real to integer conversion is not permitted in that context [Section 6]. In addition, no real to integer conversion is permitted in iteration statements (which are restricted to integer and pointer valued control variables) [Section 5.6].

B8. Type Conversion (PASCAL)

Not Satisfied.

PASCAL has implicit conversion from integer to real [p. 152] (except in for statements [p. 157] and in indexing arrays [p. 147]), no fixed point data [p. 93,143], and none of the explicit conversion operations required here. However, no conversion is necessary if an actual parameter is a variant of the corresponding formal's record type. On input, "data transfer is accompanied by an implicit data conversion operation" [p. 164].

In comparing packed arrays of characters (i.e., strings), it is not specified whether the arrays must be of the same length [p. 151].

B8. Type Conversion (SIMULA 67)

Not Satisfied.

There are no explicit conversion operators for arithmetic data. Integer-to-real and real-to-integer conversions are effected by assignment [CBL 6.1.2.1, B 80]. The function procedures rank and char provide a rudimentary explicit mechanism (i.e., at the individual character level) for converting between the object representations of numbers and their representions as characters. An actual parameter must be of a "compatible type" with its corresponding formal [B 103]. This effectively includes an actual being a constituent of a formal union type [CBL 6.1.2.2, B 151].

SIMULA incorporates the Algol rule that real expressions used as array subscripts are implicitly converted to integer values [CBL 7.1.1, AR 3.1.1]. In addition, implicit conversion between real and integer values is permitted in the evaluation of for loops [CBL 6.2].

B8. Type Conversion (TACPOL)

Partially Satisfied.

TACPOL does not allow implicit type conversions. In general, "the type of data in expressions may not be mixed" [Section 6-2]. Built-in type converters are called "Redefinition Attribute Procedures" [Section A-8]. Naturally enough, they correspond to the elementary types SHORT, LONG, CHAR, and BIT. CHAR, for example, converts object representations of numeric values to their representations as character strings. The built-in SCALE procedure [Section A-2, A-3] is used to rescale numeric values. Though not explicitly stated, the text and examples imply that parameters passed by reference must match exactly in the quantities they specify (i.e, the storage layout and its exact breakdown into elementary types) [Section 11-2]. All of the above TACPOL features satisfy the B8 requirements. However, the overlay capability of CELLs [Section 4-7] permits implicit type transfers. The example depicted in Figure 4-5 illustrates this. Moreover, the MOVE assignment operator does not check that the type of the composite object being copied is compatible with the type of the object receiving the copy.

B9. Changes in Numeric Representation (TINMAN)

Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when *any integer or fixed point value is truncated.*

(This requirement is optional for hardware installations which do not have overflow detection).

B9. Changes in Numeric Representation (ALGOL 68)

Not Satisfied.

Algol 68 does not have numerical subranges. It has no fixed point type. It does not define any action for overflow/truncation exceptions. Nor is there any exception condition available to the programmer for defining his own action.

B9. Changes in Numeric Representation (J3B)

Partially Satisfied.

J3B does not require range declarations for numeric data, but for integer variables, it does require the programmer to specify the number of bits occupied by valid data values. In addition, it supports an unsigned integer data type whose values are always supposed to be non-negative. Specifying the number of bits occupied by valid data values provides an implicit range declaration for signed integers of -2**N+1 to 2**N-1, and for unsigned integers, 0 to 2**N-1 [Section 1.3.3.1]. These implicit "range" declarations

are not checked at compile time when assigning to integer variables [Section 5.2], thus satisfying B9's requirement for no "explicit conversion operations ... between numeric ranges." On the other hand, if a formal parameter is declared to be S 5, for example, then the actual parameter must also be an S 5 [Section 5.3.3]. In effect, the implicitly declared "ranges" must match exactly. The programmer can deactivate this matching requirement, however, by specifying the formal to be of type S (without any implicit "range"), to imply that the actual parameter may be any integer value [Sections 4.8 and 5.3.3].

Range validation at run-time is not under programmer control within the language. In addition, J3B provides no exception condition for integer or fixed point overflow occuring as the result of either an arithmetic opration or assignment.

B9. Changes in Numeric Representation (PASCAL)

Partially Satisfied.

PASCAL supports implicit conversion from integer to real and from compatible subranges [p. 152] but does not support fixed point numbers [p. 93]. PASCAL 6000 has run time tests for range errors as a user option [p. 101] with error messsage "element expression out of range" [p. 121]. There is not, however, an exception handling facility in PASCAL.

B9. Changes in Numeric Representations (SIMULA 67)

Not Satisfied.

There is no run time exception condition for numeric range errors. Rather, the prevention and/or detection of conditions such as overflow are the programmer's responsibility. To do this one must take the specific hardware representations into account [CBL 4.1.1, AR 3.3.6].

B9. Changes in Numeric Representation (TACPOL)

Satisfied.

TACPOL provides a FOFL condition for fixed overflow [Section 12-1]. If the programmer declares the FOFL condition with CHECK, a snapshot procedure is invoked.

B10. Io Operations (TINMAN)

The base language will provide operations allowing programs to interact with files, channels, or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

I/O operations can be made accessible in a HOL in an abstract form through a small set of generic I/O operations (like "read" and "write, " with appropriate device and exception parameters). Note that devices and files are similar in many respects to types, so additional language features may not be required to satisfy this requirement. This requirement, in conjunction with requirement E1, permits user definition of unique equipment and its associated I/O operations as data types within the syntactic and semantic framework provided by the generic operations.

B10. I/O Operations (ALGOL 68)

Satisfied.

Algol 68 has procedures for dynamically associating internal file variables with external data sets and channels/devices [Section 10.3.1.4]. It has procedures for formatless [Section 10.3.3], formatted [Section 10.3.5], and binary [Section 10.3.6] I/O. Support is provided for simultaneous read access by several processes [Section 10.3.1.1(bb)], as well as user control of some exception conditions [Section 10.3.1.3.cc], e.g. end of file.

Formatted I/O provides more elaborate editing capabilities that B10 requires, but Algol 68's I/O capabilities in other respects, however, may provide a useful basis for satisfying B10.

B10. I/O Operations (J3B)

Not Satisfied.

J3B provides no input/output operations.

B10. I/O Operations (PASCAL)

Partially Satisfied.

The data type "file" with associated operations get, put, reset, and rewrite is supported [p. 55]. The correspondence between a program variable of type file and a peripheral device or logical file is implementation dependent and, so, outside of standard PASCAL [p. 55]. External files are made available "as actual parameters in the program call statement" [p. 91]. These are then established for the duration of program execution. Therefore, programs cannot dynamically assign and reassign I/O devices. Only the exception condition end-of-file, plus end-of-line for the line-oriented textfiles (i.e., file with component type "char") are provided for user control [p. 161, 165].

B10. I/O Operations (SIMULA 67)

Satisfied.

The class FILE is defined (as a subclass of BASICIO) with four subclasses predefined: infile, outfile, directfile, and printfile. A subclass of FILE can include implementation code to communicate data with an external device (CL 11, 11.1). A subclass parameter "ties the internal file object to the external medium" [B 192]. Exception conditions (e.g., more, endfile, lastitem) are available, or can be coded as Boolean procedures, for user control.

Given encompassing declarations such as:

```
BEGIN CLASS file ...;
          file CLASS device1 (-) ...;
          file CLASS device2 (-) ...;
          REF (file) source;
```

Then the code fragments

```
source :- NEW device1 (-);
          .
          .
          .
source :- NEW device2 (-);
```

suggest how "to dynamically assign and reassign I/O devices" in SIMULA 67 [CBL 11.-entire section, B 190-207].

SIMULA 67's I/O capabilities may be a good example of how B10 should be satisfied.

B10. I/O Operations (TACPOL)

Partially Satisfied.

TACPOL provides for input/output via READ [Section 13-6] and WRITE [Section 13-7] statements operating on declared FILEs [Section 14-3]. The user can control the subsequent actions on three file processing exception conditions using the ON statement [Section 13-13]. These are ENDFILE [Section 13-13.a] for serial input files, NOKEY [Section 13-13.b] for a key error in a direct access file, and NOPART [Section 13-13.c] for determining "whether or not an attempt was made to open an old partition in a file when that partition no longer exists." It appears that TACPOL programs cannot "dynamically assign and reassign I/O devices" since "file declarations may not appear outside the Compool" [Section 14-1]. Moreover, "the Compool is like a block in which all programs are contained" [Section 14-2]. This means only a single device can be associated with a given program identifier (declared to be a file name) during a program's execution.

TACPOL requires support for partitioned data sets and file labels [Section 14-3]. This is more than a minimal generic set of I/O concepts.

B11. Power Set Operations (TINMAN)

The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

B11. Power Set Operations (ALGOL 68)

Not Satisfied.

Algol 68 does not have enumeration types. It does not have power set types. It does have a bits type [Section 10.2.2.g] that is an implementation dependent fixed size array of Booleans. This provides for sets of integers. There are union, intersection, complement, and element operators [Section 10.2.3.8] for bits data and operators for converting between bits and "array of Boolean" types. Shift (left and right) operations are also provided. No operator corresponding to set difference is pre-defined, however.

B11. Power Set Operations (J3B)

Not Satisfied.

J3B does not support the concept of enumeration types nor power sets. It does provide a bit string data type, however, with AND, OR, XOR, NOT, and a bit extraction operator that can be used to test whether a given element in the string is a TRUE or FALSE value [Section 9.2]. An operation corresponding to set difference is not part of the language, however.

B11. Power Set Operations (PASCAL)

Partially Satisfied.

PASCAL has a type "set" which defines "the powerset of its so-called base type" [p. 144]. Enumeration types are subsumed as permissible base types [pp. 144, 142]. Union, set difference, intersection, and a membership predicate are built-in operators applicable to all set types [p. 145]. A complement operator is not included.

B11. Power Set Operations (SIMULA 67)

Not Satisfied.

SIMULA 67 does not offer "data types defined as power sets of enumeration types" since it does not even offer enumeration tyes. For, outside of object reference types, SIMULA 67 has only real, integer, Boolean, array, character, and text types [CBL 3.1, AR 5.1.1].

B11. Power Set Operations (TACPOL)

Not Satisfied.

As mentioned in A5, TACPOL does not support enumeration types, so power sets of such types and the associated set operations are absent from the language. However, TACPOL does have a bit string data type [Section 3-3.b] that can directly represent a set. The logical operators AND, OR, and NOT [Section 6-1.d] then realize union, intersection, and complement respectively. The substring operation SUBSTR [Section 6-1.c.(2)] applied to a bit string suffices as an element predicate. No operation equivalent to set difference is built-in, however.

C1. Side Effects (Due to Evaluation Order) (TINMAN)

Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

This provides a simple rule (i.e., left-to-right) for order of argument evaluation, but allows implementations to alter the order in any way which does not change the effect.

C1. Side Effects (Due to Evaluation Order) (ALGOL 68)

Not Satisfied.

In Algol 68, operands [Section 5.4.2], arguments [Section 5.4.3], and subscripts [Section 5.3.2] are evaluated collaterally, i.e., the order of evaluation is implementation dependent.

C1. Side Effects (Due to Evaluation Order) (J3B)

Not Satisfied.

J3B explicitly does not require that array indices or operands of expressions be evaluated in left-to-right order [Section 7.1, p. 7-3 and Section 5.2, p. 5-3]. Nor is it required that the target of an assignment statement be evaluated before the value to be assigned is determined.

C1. Side Effects (Due to Evaluation Order) (PASCAL)

Not Satisfied.

In expression evaluation "sequences of operators of the same precedence are executed from left to right" [p. 148], i.e., operand evaluation order is not defined. The evaluation order of subscripts [p. 147] and parameters [pp. 151, 152-153] is not defined, nor is the target of an assignment statement specified to be evaluated before the value to be assigned is determined.

C1. Side Effects (Due to Evaluation Order) (SIMULA 67)

Not Satisfied.

SIMULA 67 does not constrain the order of argument evaluation for an operator. This leaves a major ambiguity about side effects in the language which it inherits from ALGOL 60 since SIMULA 67 subsumes ALGOL 60's syntax and semantics for arithmetic expressions [CBL 4.1.1].

C1. Side Effects (Due to Evaluation Order) (TACPOL)

Not Satisfied.

Argument evaluation order is not specified for TACPOL, although "expressions are evaluated from left to right according to the priorities of the operators" [Section 6-2.a]. The explanatory examples seem to imply a left-to-right operand evaluation order.

C2. Operand Structure (TINMAN)

Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

Care must be taken to ensure that the operator/operand structure of expressions is not psychologically ambiguous (i.e., to guarantee that the parse implemented by the language is the same as intended by the programmer and understood by those reading the program). This kind of problem can be minimized by having few precedence levels and parsing rules, by allowing explicit parentheses to specify the intended execution order, and by requiring explicit parentheses when the execution order is of significance to the result within the same precedence level (e.g., "X divided by Y divided by Z" and "X divided by Y multiplied by Z"). The user will not be able to define new operator precedence rules nor change the precedence of existing operators.

C2. Operand Structure (ALGOL 68)

Not Satisfied.

Algol 68 has ten precedence levels (monadic plus nine dyadic) [Section 4.3]. The precedence of standard operators is essentially conventional [Section 10.2.3.0]. For expressions involving operators at the same level, explicit parenthesis may be used to force the association of operands with operators. In their absence, the default is left to right [Section 5.4.2], e.g. (A+B)+C. The programmer may define (redefine) the precedence of new (existing) operators [Section 4.3].

C2. Operand Structure (J3B)

Partially Satisfied.

Arithmetic operations and logical operations satisfy conventional precedence relationships [Sections 7.1 and 7.3]. Explicit parentheses are permitted to specify the intended grouping of operands with operators. But explicit parentheses are not required when the grouping of operands with operators is of significance to the result within the same precedence level [Section 7.1.1-4].

A programmer cannot redefine the precedence of existing operators, nor can new operator precedence rules be introduced.

C2. Operand Structure (PASCAL)

Partially Satisfied.

PASCAL has only 4 levels of operator precedence [p. 149] - not, multiplication, addition, and relational operators - in contrast to ALGOL 60's 9 levels. In retrospect, Wirth believes that this simplification may break too much from tradition [Wirth, p. 194]. Explicit parentheses are not required within the same precedence level. A programmer cannot redefine the precedence of existing operators, nor can new operator precedence rules be introduced.

C2. Operand Structure (SIMULA 67)

Partially Satisfied.

SIMULA 67 has nine levels of operator hierarchy from ALGOL 60 [AR 3.4.6.1, 3.3.5.1] and it does not require explicit parentheses when the execution order is of significance to the result within the same precedence level because "can" [AR 3.3.5.2] rather than "must" qualifies the use of parentheses in such situations.

C2. Operand Structure (TACPOL)

Partially Satisfied.

TACPOL has a standard operator hierarchy [Section 6-2]. The user cannot define new operator precedence rules nor change the precedence of existing operators. However, TACPOL does not require explicit parentheses when the execution order is of significance to the result within the same precedence level. Here, left-to-right evaluation is the determining rule.

C3. Expressions Permitted (TINMAN)

Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

FORTRAN, for example, has a list of seven different syntactic forms for subscript expressions, instead of allowing all forms of arithmetic expressions.

C3. Expressions Permitted (ALGOL 68)

Satisfied.

This can be verified by checking the syntax. One of Algol 68's design goals is orthogonality [Section 0.1.2].

C3. Expressions Permitted (J3B)

Satisfied.

In general, wherever a variable is to be evaluated for its value at run time, the syntax of J3B permits substituting a constant or expression. There are three exceptions, however. The INTR, BYTE, and BIT functional modifiers [Section 9.2] accept only variables as their arguments. This violates the spirit, if not the letter of C3, since both variables and constants are not permitted as arguments of these functions.

C3. Expressions Permitted (PASCAL)

Satisfied.

This can be verified directly from the formal specification of the language's syntax [pp. 110-118].

C3. Expressions Permitted (SIMULA 67)

Satisfied.

Examination of the defining syntax [CBL] shows constant to be only instances of expressions; they are not used elsewhere.

C3. Expressions Permitted (TACPOL)

Not Determinable.

The Reference Manual is not sufficiently explicit to evaluate TACPOL fully with respect to C3. Expressions are permitted for initial, increment, and final values of iterative DOs [Section 8-3.d]. They also may appear as procedure VALUE arguments [Section 11-3] and as the RETURN value for function procedures [Section 9-3]. This seems to be where both constants and variable references are permitted.

Exponentiation is restricted to positive integer constants, but since variables and expressions containing variables are excluded, this does not violate the letter of C3, although it perhaps violates its spirit.

The SUBSTR operator is described as requiring a string valued identifier [Section 6-1.c.(2)] as its first argument rather than a constant or expression. This does not violate the letter of C3, though it does violate its intent.

C4. Constant Expressions (TINMAN)

Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.

The resulting value should be the same (at least within the stated precision) regardless of the object machine (see D2). Allowing constant expressions in place of constants can improve the clarity, correctness, and maintainability of programs and does not impose any run time costs.

C4. Constant Expressions (ALGOL 68)

Not Satisfied.

Algol 68 allows constant expressions wherever constants are allowed, except in priority declarations [Section 4.3]. However, it does not require them to be evaluated at compile time.

C4. Constant Expressions (J3B)

Partially Satisfied.

The J3B language specification explicitly describes which expressions consisting entirely of constant values will be evaluated at compile time [Section 8]. However, function calls whose arguments are all constant, the absolute value function, the INTR, BYTE, BIT, SHIFTR, and SHIFTL functional modifiers, exponentiation, and relational formulas are never evaluated at compile time. In addition, expressions of the form TRUE OR VAR or FALSE AND VAR (where VAR is a variable or expression) are not evaluated at compile time. Of these restrictions, the most significant is the lack of compile time evaluation of relational expressions, since this limits J3B's conditional compilation capability.

Numeric expressions are specified to be evaluated with host rather than object machine range and precision [Section 8.1.2 and 8.1.3].

C4. Constant Expressions (PASCAL)

Not Satisfied.

PASCAL does not permit constant expressions in the following four contexts:

(1) constant definition part

CONST <identifier> = <constant>; [pp. 110, 118, 159]

(2) CASE labels [pp. 114, 112, 118]

(3) Defining subrange types (i.e., the lower and upper bounds for possible scalar values) <constant> .. <constant> e.g. pp. 111, 116, 143)

(4) GOTO <label> where label is an unsigned integer constant [pp. 110, 153].

C4. Constant Expressions (SIMULA 67)

Not Satisfied.

Constant expressions are allowed anywhere constants are allowed but are not required to be "evaluated before run time".

C4. Constant Expressions (TACPOL)

Not Satisfied.

Some contexts are apparently restricted to constants (rather than constant expressions). For example, all illustrations of declaring array upper bounds [Section 4-2] use only integer constants, although no prohibition of constant expressions is actually stated. In addition, precision and scale specifications must be "numbers" [Section 4-1.a]. Program constants declared in "Value declarations" [Section 4-9] require literal constants in their associated INIT attribute where their values are given.

C5. Consistent Parameter Rules (TINMAN)

There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handing, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters.

C5. Consistent Parameter Rules (ALGOL 68)

Partially Satisfied.

Algol 68 does not have exception conditions. Its parallel processes are parameterless, as are its data types. The parameter rules are consistent for operators and procedures [Section 5.4.2, 5.4.3]. However, for arrays there is a special subscripting operation for selecting subarrays [Section 5.3.2], e.g., A[2:4, 3:5].

C5. Consistent Parameter Rules (J3B)

Partially Satisfied.

Although J3B distinguishes read-only from writable parameters in the form of a subroutine call [Section 5.3], the specified syntax is such that array and table indexing is consistent with the form of a subroutine call, since the indices are not writable "parameters". Moreover, consistency is further supported in that functions are permitted to have writable parameters [Section 3.3.3]. However, the use of a colon to separate read-only from writable parameters is not consistent with its use to separate lower and upper bounds of table indices in declarations [Section 4.3] nor is it consistent with the use of POINT (table name, from : to) in table initialization [Section 4.9].

The declaration of subroutines (as opposed to their definition) permits a special type to be specified as a formal parameter to indicate that an integer, bit, or character string argument of any size is deemed to match the corresponding formal parameter. For example,

```
PROC F(S);
```

declares F to be a procedure accepting integer-valued arguments of any size, even though when defining F,

```
PROC F(II); BEGIN
   ITEM II S 31;
```

the formal parameter II must be declared to be of a specific size [Section 4.8].

J3B has no exception handling capability, parallel process control operations, or parameterized type declarations, so C5 is vacuously satisfied for these capabilities.

The BIT and BYTE pseudo variable syntax, however, forms an exception to the parameter notational conventions. In the assignment statement:

```
BIT(STRING, II, 1) = BIT.VAR;
```

the value of the variable "STRING" is modified. To be consistent with the parameter notations elsewhere in the language, the syntax of the BIT and BYTE pseudo variables should require a colon before the first argument, to show that STRING (at least) is a writable parameter.

Another exception to parameterization rules is that ordinary tables that are not declared as "typed tables" may not be passed as parameters (p. 5-12). This is inconsistent with the general ability to pass tables as parameters. However, the provision of untyped ordinary tables conflicts with other TINMAN requirements, so this inconsistency vanishes if J3B is brought into conformity with other TINMAN requirements.

The form and interpretation of actual macro parameters differs from that of subroutine actual parameters (compare Sections 2.2.6 and 3.4 with 3.3.2, 3.3.3, and 5.3). In addition, formal macro parameters are not declared to be of any particular data type.

C5. Consistent Parameter Rules (PASCAL)

Partially Satisfied.

PASCAL does not permit parameterization of type declaration and use; it does not provide for exception handling or parallel processing. The language is consistent in its parameter rules for procedures and functions [pp. 158, 162] in that functions may have output parameters. However, "procedures and functions which are used as parameters to other procedures and functions must have value parameters only" [p. 158].

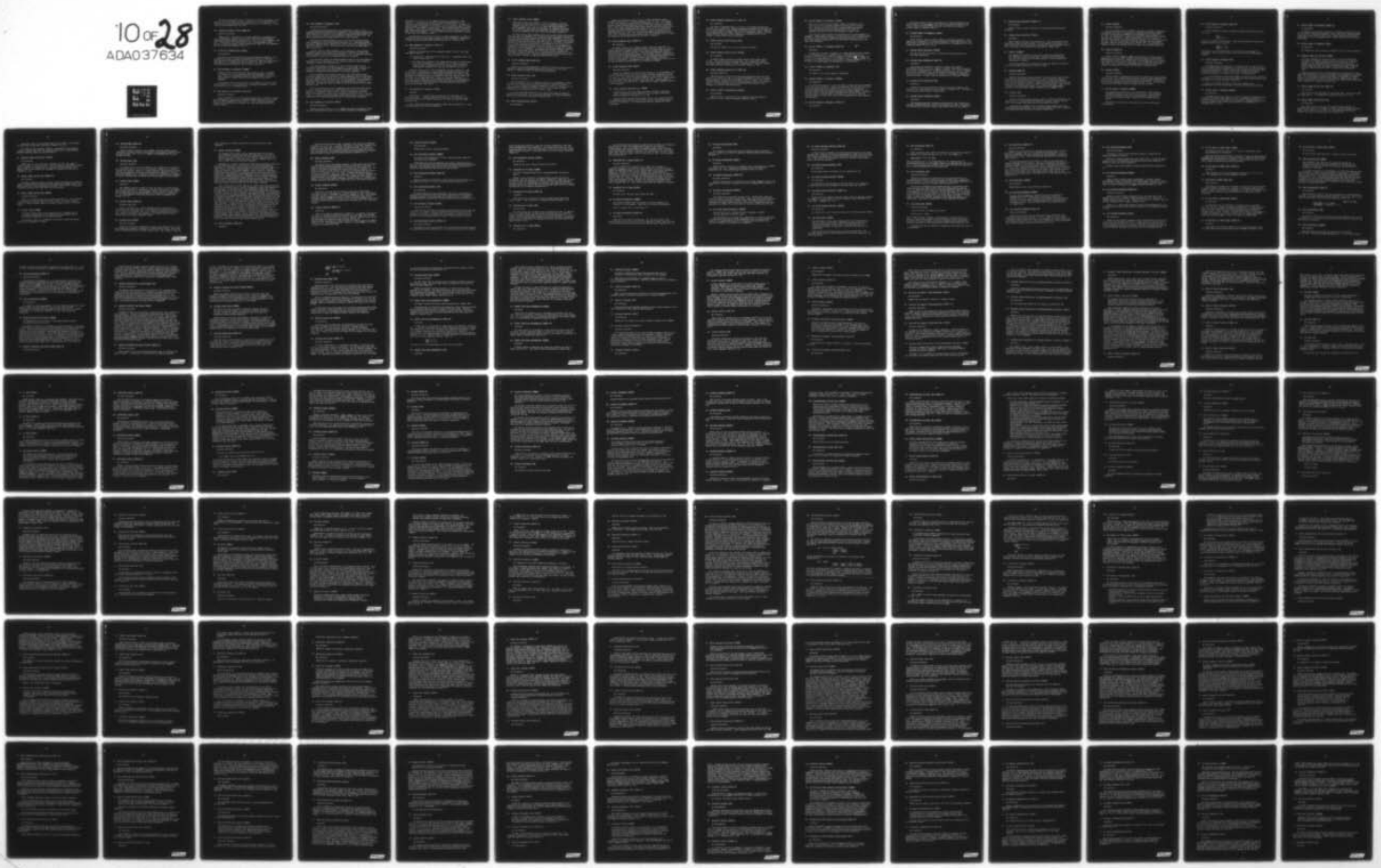
The standard arithmetic functions abs and sqr are generic [p. 163] (the type of the argument determines the type of result), but a programmer cannot define generic subroutines.

Input and output procedures for textfiles, use "a non-standard syntax for their parameter lists, allowing, among other things, for a variable number of parameters" [p. 164] and for groups of parameters separated by commas. For addressing efficiency reasons the restriction that "components of a packed structure must not appear as actual variable parameters" [p. 53] is imposed.

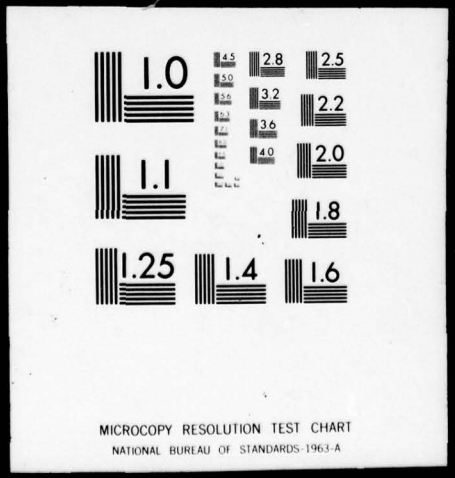
AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

10 OF 28
ADA037634



10 OF 28
ADA037634



The form of declaration used for subroutines passed as parameters is the same as the form used when defining a subroutine, i.e., in both cases the parameter types are specified by declaring the type of an identifier serving as the formal parameter [pp. 158, 162].

C5. Consistent Parameter Rules (SIMULA 67)

Partially Satisfied.

SIMULA 67 has a consistent set of rules applicable to parameters and their transmission. However, only a subset of the procedure parameter transmission capabilities are available for class (i.e., type) parameters [CBL 8.2]. Specifically call-by-name mode is not a class parameter option. Further, function procedures, labels, and switches cannot be class parameters but can be procedure parameters [CBL 8.2].

C5. Consistent Parameter Rules (TACPOL)

Not Satisfied.

Procedures and functions satisfy the same parameter transmission rules, including the ability to assign to parameters of functions. Built-in functions have some capabilities denied user-defined functions, specifically, some arguments are optional and some functions, e.g. ABS, are generic. Subroutines passed as parameters may not take any arguments.

C6. Type Agreement in Parameters (TINMAN)

Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

Some notations permit such parameters to be implicit on the call side. Formal parameters of a union type will be considered conformable to actual parameters of any of the component types.

C6. Type Agreement in Parameters (ALGOL 68)

Partially Satisfied.

Algol 68 satisfies all the requirements except that it allows an integer actual parameter to be implicitly converted (widened) [Section 6.5] to real when the formal parameter is real [Section 5.4.3]. Implicit conversions between reals of different precisions, e.g., real and long real, are not allowed, however.

C6. Type Agreement in Parameters (J3B)

Partially Satisfied.

Integer-valued formal parameters can be specified to match actual parameters exactly with respect to the parameter's size declaration (where the "size" specifies the number of bits permitted to hold valid data values) or to ignore the size attribute of actual parameters [Section 4.8 and 5.3.3]. The actual parameter must be integer-valued in all cases, however.

Floating point formal parameters must be matched with actual floating point parameters of the same precision [Section 5.3.3]. Fixed point formal parameters similarly must be matched with actual fixed point parameters of the same scale [Section 5.3.3]. (All fixed point values have the same number of bits of precision.)

Character and bit formal parameters can be specified to match actual parameters exactly with respect to the parameter's length or to ignore the length attribute of actual parameters [Section 5.3.3]. No implicit conversion from character to bit or bit to character values is permitted.

Array and table formal and actual parameters must have the same number of dimensions but, in violation of C6, the subscript range for these parameters is fixed at compile time [Section 4.8].

J3B's type matching constraints for J3B's "typed" tables in effect satisfy C6's constraints on formal parameters of a union type, since the J3B constraints ensure that every record field accesssible through a formal parameter is supported by the structure of the actual parameter.

C6's constraints are violated, however, by J3B's "specified" tables, since when a specified table serves as a formal parameter, the only constraint an actual parameter must satisfy is that the actual parameter be a table with the same number of words per entry as the formal parameter, the same number of dimensions, and the same subscript range (if subscripts are permitted). In particular, the component types of formal and actual parameters need not match [Section 5.3.3].

C6 is also violated by pointer parameters, which may be declared untyped as formal parameters [Section 4.8], meaning that no type check on actual parameters is performed [Section 5.3.3]. Typed formal pointer parameters, however, are checked in a manner that satisfies C6 [Section 5.3.3]. i.e., the fields of a record accessible through a typed formal pointer parameter must be supported by the record accessible through the actual pointer parameter.

C6. Type Agreement in Parameters (PASCAL)

Partially Satisfied.

Though not stated explicitly in the PASCAL User Manual and Report, formal and actual parameters must agree in type, where a formal of a union type is

conformable to an actual of any component type as in assignment. This inference is justified both by current PASCAL implementations and the discussion of formal/actual correspondence [ 711-72 152-253]. In PASCAL, not only the number of dimensions but also the size and subscript range for array parameters is determined at compile-time. Since formal procedure parameters must specify their type for exact matching, and since the size is part of the type of an array parameter, "a given procedure can only be applied to arrays of one fixed size" [Wirth, p. 194].

For procedures and functions declared as formal parameters, no method is provided for declaring the type and number of their parameters; consequently no type checking is possible when such subroutines are called [p. 79].

C6. Type Agreement in Parameters (SIMULA 67)

Partially Satisfied.

Formals and actuals must "be of compatible types" [B 103]. This means they agree in type or

(1) the actual is a subclass of the formal (i.e., a component type of the formal's union type), or

(2) the formal is a subclass of the actual and the current actual's value is in that subclass (i.e., the actual is of a union type and its current value is "compatible" with the formal's type). Obviously "this must be checked at run time" [B 151] [CBL 8.2, 6.1.2.2]

SIMULA 67 fully satisfies C6's requirements for array parameters. That is, the number of dimensions is compile time determinable and the subscript ranges must be passed as part of the procedure arguments [B 101-105]. Type agreement for procedures passed as parameters means only that the formal and the actual are both procedures [AR 5.4.1], i.e., no type checking is prescribed on the number and type of arguments permitted. Type agreement on functions passed as parameters requires only that "the type associated with the actual parameter must coincide with or be subordinate to that of the formal specification" [CBL 8.2.2].

C6. Type Agreement in Parameters (TACPOL)

Satisfied.

"The parameter attributes must correspond to the attributes of the argument by type ... but not necessarily by size or scale" [Section 11-3]. It is implied that array formals and actuals must match in subscript range [Section 11-2].

Since subroutines passed as parameters cannot take any arguments, no type checking problems arise for such parameters.

C7. Formal Parameter Kinds (TINMAN)

**There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.**

The first class of data parameter acts as a constant within the procedure body and cannot be assigned to nor changed during the procedure's execution; its corresponding actual parameter may be any legal expression of the desired type and will be evaluated once at the time of call. The second class of data parameter renames the actual parameter which must be a variable. The address of the actual parameter variable will be determined by (or at) the time of call and unalterable during execution of the procedure, and assignment (or reference) to the formal parameter name will assign (or access) the variable which is the actual parameter. A language with exception handling capability must have a way to pass control and related data through procedure call interfaces. Exception handling control parameters will be specified on the call side only when needed. Actual procedure parameters will be restricted to those of similar (explicit or implicit) specification parts.

C7. Formal Parameter Kinds (ALGOL 68)

Partially Satisfied.

Algol 68 has the (non-assignable) constant and variable renaming kinds of parameters, along with procedure parameters [Section 5.4.3, 4.4, 4.6]. However, it does not support exception conditions per se.

C7. Formal Parameter Kinds (J3B)

Partially Satisfied.

J3B satisfies C7 in that it supports read-only input parameters (called by value) and writable output parameters (called by reference) [Section 5.3.3]. Input parameters (including components of tables or arrays passed as input parameters) cannot be assigned to nor can they be used as output parameters in a subroutine call (p. 5-7 and p. 5-14).

C7 is not satisfied, however, in that subroutines cannot be passed as parameters [Section 4-8], nor is there any provision for exception handling. Labels cannot be passed as parameters [Section 4-8].

C7. Formal Parameter Kinds (PASCAL)

Not Satisfied.

PASCAL has essentially three classes of formal parameters: value, variable, and function or procedure parameters [pp. 71, 152]. The variable and procedure parameters satisfy the C7 requirements but there is no formal parameter class for specifying the control action when exception conditions occur. Also, value parameters receive their initial value from the actual in the call, but "the procedure may then change the value of this variable by assigning to it" [p. 72] since it is, in effect, "a local variable of the called procedure" [p. 153]. This violates the TINMAN read-only stipulation that a value parameter acts as a constant within the procedure body and cannot be assigned to nor changed during the procedure's execution.

C7. Formal Parameter Kinds (SIMULA 67)

Not Satisfied.

In SIMULA 67 there is no "formal parameter class for specifying the control action when exception conditions occur" although a label or a switch may be passed as a parameter [CBL 8.2] for control transfer upon program discovery of an exception condition. There are three parameter transmission modes: by value, by reference, by name. The "by name" mode is excluded by C7 and the "by value" mode for SIMULA 67 is not sufficiently restrictive since a by-value actual parameter can be assigned to and "changed" during the procedure's execution". Procedures can be passed as parameters [CBL 8.2], but no type checking is possible when the procedures are invoked.

C7. Formal Parameter Kinds (TACPOL)

Partially Satisfied.

TACPOL has parameter call by reference [Section 11-2] and parameter call by value [Section 11-3], along with procedure and label parameters [Section 11-4, 11-5]. "Any changes to a value parameter in the invoked procedure do not affect the value in the invoking procedure" [Section 11-3]. This violates C7's requirement that value parameters are read only. Label parameters [Section 11-5] are excluded by C7. TACPOL does not support exception handling control parameters.

C8. Formal Parameter Specifications (TINMAN)

> Specification of the type, range, precision, dimension, scale and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.

Optional formal parameter specification permits the writing of generic procedures which are instantiated at compile time by the characteristics of their actual parameters. It eliminates the need for compile time "type" parameters.

C8. Formal Parameter Specifications (ALGOL 68)

Not Satisfied.

The types of parameters must be specified for procedure declarations [Section 5.4.1] in Algol 68. However, operators are generic in the sense that one can have different routines associated with an operator token, depending on the operand types, and these routines may be programmer defined. But this limits ALGOL 68's generic routine capability to routines of one or two arguments invoked syntactically as operators.

C8. Formal Parameter Specifications (J3B)

Not Satisfied.

J3B does not support the concept of generic procedures.

C8. Formal Parameter Specifications (PASCAL)

Not Satisfied.

In PASCAL "formal procedure parameters specify their types" [Wirth, p. 194]. The specification is not optional. As a consequence, generic procedures cannot be written in PASCAL. Admittedly this "seriously impairs the highly desirable flexibility of procedures" [Wirth, p. 194].

C8. Formal Parameter Specification (SIMULA 67)

Partially Satisfied.

"Specification is required for each formal parameter" [CBL 8.2] where specification "gives the type of each formal parameter" [B 103]. Procedures that are generic in more than one argument cannot be written. The virtual concept, however, provides a form of generic procedure [CBL 2.2.3, B 154-161] wherein a procedure identifier can be associated with procedure bodies belonging to different type definitions.

C8. Formal Parameter Specifications (TACPOL)

Not Satisfied.

TACPOL requires specification of all parameter types and dimensions [Section 11-2, 11-3]. Generic procedures cannot be written.

C9. Variable Numbers of Parameters (TINMAN)

There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

There are many useful purposes for procedures with variable numbers of arguments. These include intrinsic functions such as "print," generalizations of operations which are both commutative and associative such as "max" and "min," and repetitive application of the same binary operation such as the Lisp "list" operation.

C9. Variable Numbers of Parameters (ALGOL 68)

Satisfied.

In Algol 68, the number of parameters is fixed [Section 5.4.3]. However, an actual parameter may be a flexible array. The Algol 68 row display [Section 3.3] allows one to explicitly specify the elements of an array. e.g. (1, 2, 3) is a three element array of integers. Thus, a variable number of parameters is obtained by using a row display as an array parameter, e.g. get (f, (X, Y, Z)) reads values for variables X, Y, Z from file f.

C9. Variable Numbers of Parameters (J3B)

Not Satisfied.

J3B supports only a fixed number of parameters.

C9. Variable Numbers of Parameters (PASCAL)

Not Satisfied.

In PASCAL there must be an exact match both in number and in type between actual and formal parameters. "Upon an activation of the procedure statement, an actual quantity has to be indicated for each parameter which can be referenced from within the procedure through the formal parameter" [p. 13)]. Only the standard input and output procedures read, write, readln, writeln for textfiles allow a variable number of parameters [p. 164-166]. And this variable number is in each case compile-time determinable from the individual call statements.

C9. Variable Number of Parameters (SIMULA 67)

Not Satisfied.

The actual parameter list in a procedure call "must correspond to the formal parameter list in number, order and be of compatible types" [B 103]. In fact, even the standard input and output procedures do not permit a variable number of arguments [CBL 11].

C9. Variable Number of Parameters (TACPOL)

Not Satisfied.

TACPOL does not support variable numbers of procedure parameters. "The argument list appearing in the invoking statement or expression must have the same number of arguments as there are identifiers in the parameter list of the invoked procedure" [Section 11-1.a].

D1. Constant Value Identifiers (TINMAN)

The user will have the ability to associate constant values of any type with identifiers.

D1. Constant Value Identifiers (ALGOL 68)

Satisfied.

Identifiers are declared as constants in Algol 68 by identity declarations [Section 4.4], e.g., real pi = 3.14159. Algol 68 also permits the run time initialization of a constant identifier, i.e., such a "constant" may be given a different value (by an identity declaration) on each entry to the scope in which it is declared, but it may not otherwise be assigned to within its scope. In effect, such an identifier is a "read-only" variable.

D1. Constant Value Identifiers (J3B)

Satisfied.

Constant values of any scalar data type can be given a mnemonic name [Section 4.7]. Such names may not be assigned to [Section 5.2]. The value of such constants must be known at compile time.

D1. Constant Value Identifiers (PASCAL)

Satisfied.

Each procedure may have a constant definition part that "contains all constant synonym definitions local to the procedure" [p. 159]. The value of the constant must be known at compile time (unlike ALGOL 68, for example).

D1. Constant Value Identifiers (SIMULA 67)

Not Satisfied.

SIMULA 67 does not have a facility for letting identifiers represent constants.

D1. Constant Value Identifiers (TACPOL)

Satisfied.

TACPOL supports constant value identifiers. They are declared in value declarations using the particle INIT. "Once a name has been declared in a value declaration, the value assigned that name may never be changed" [Section 4-9]. The value of such constants is determined at compile time.

D2. Literals (TINMAN)

The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).

Literals are needed for all atomic data types and should be provided as part of the language definition for built-in types.

D2. Literals (ALGOL 68)

Partially Satisfied.

Algol 68 denotations provide constants for the primitive types: integer, real, Boolean, and character [Section 8.1]. Row and structure displays [Section 3.3] provide array and record constants. Routine texts [Section 5.4.1] provide procedure constants. However, pointer constants other than nil depend on the compiler's ability to detect constant addresses. Algol 68 does not satisfy the requirements because it does not require real program constants to yield the same internal representation as real I/O data.

D2. Literals (J3B)

Partially Satisfied.

Integer, floating, fixed, character, and bit literals are defined in J3B [Section 2.2.3]. The value NULL is defined as a pointer constant [Section 8.4]. So all atomic data types have literals defined for them.

J3B provides no capability for converting between numeric values and their printable representation, and so does not address the issue of ensuring that program and data literals are converted identically.

D2. Literals (PASCAL)

Partially Satisfied.

If v is an integer or real variable, then read (v) invokes reading from the standard input file "of a sequence of characters which form a number according to the syntax of PASCAL and the assignment of that number to v" [p. 165]. No constraints are imposed to ensure the program and data literals are converted identically. Similar remarks apply if v is of type char. Boolean, real, integer, char and even string constants can be written directly [p. 166]. However, no provision is made for the input (via the standard procedure read) of Boolean constants [pp. 84-85, 165]. No constraints are imposed to ensure that program and data literals are converted identically.

D2. Literals (SIMULA 67)

Partially Satisfied.

Boolean constant (FALSE and TRUE) and the no-object reference constant NONE cannot be read as input constants or written except as TEXT values, although they can appear as constants within the program code. For example, only character, text, integer, fixed point, or real constants may be read as input [CBL 11.2]. Further, the programmer cannot specify or control the accuracy of conversion of literals from external to internal form.

D2. Literals (TACPOL)

Partially Satisfied.

TACPOL provides a straightforward syntax for numeric and string constants [Section 3-5]. The reference manual does not detail the input representation for constants [Section 13-6], but one wonders if long numeric inputs have a final L as do long numeric program literals [Section 3-5.6]. No restriction is stated on conversion accuracy of program and data literals.

D3. Initial Values of Variables (TINMAN)

The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

There will be provision (at user option) for run time testing for initialization.

D3. Initial Values of Variables (ALGOL 68)

Partially Satisfied.

Initial values are optional in Algol 68 variable declarations [Section 4.4], e.g.

```
int i;
real x := 0;
```

There are no default initial values. Algol 68 declarations are executed in sequence [Section 3.2], so that in

```
real X :=  Y;
real Y := 3.5;
```

the value of Y is not defined when X is initialized, even though Y is accessible. Algol 68 does not define any run time test for uninitialized variables.

D3. Initial Values of Variables (J3B)

Partially Satisfied.

Programmer-defined initialization is supported only for data allocated prior to program execution [Sections 4.4, 4.5, and 4.9]. If such data are not explicitly initialized, a default initial value represented by all zero bits is provided [Sections 6, p. 6-3], even if such a value is not legally permitted according to a variable's type [e.g., see Section 8.4].

It is not possible to initialize variables declared local to a subroutine [Sections 4.3, 4.4, and 4.5], and typed tables can only be initialized if they are declared in a block [Sections 4.3 and 4.1].

There is no support provided in either the language or the implementations for run time testing of access to uninitialized variables.

D3. Initial Values of Variables (PASCAL)

Not Satisfied.

"Variable declarations consist of a list of identifiers denoting the new variables, followed by their type" [p. 146]. No mechanism or syntax for initialization in the declaration part is provided. Consequently for locally declared variables "their values are undefined at the beginning of the statement part" [p. 159].

D3. Initial Values for Variables (SIMULA 67)

Not Satisfied.

In SIMULA 67 "any declared variable is initialized at the time of entry into the block to which the variable is local. The initial contents depends on the type of the variable" [CBL 3.2.4]. Default initial values violate requirement D3, and no programmer-defined initial values can be specified in declarations.

D3. Initial Values of Variables (TACPOL)

Not Satisfied.

The TACPOL user cannot initialize variables as part of their declaration.

D4. Numerical Range and Step Size (TINMAN)

> The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

Range specifications offer the opportunity for the translator to insert range tests automatically for run time or debug time validation of the program logic. With the ranges of variables specified in the program, it becomes possible to perform many subscript bounds checks at compile time. These bounds checks, however, can be only as valid as the range specifications which cannot in general be validated at compile time.

D4. Numeric Range and Step Size (ALGOL 68)

Not Satisfied.

Algol 68 has neither subranges nor fixed point type. The range of real, int, long real etc. is implementation dependent [Section 2.1.3.1].

D4. Numeric Range and Step Size (J3B)

Not Satisfied.

The range of integer data can only be specified approximately by declaring a variable to be a signed or unsigned integer occupying a specific number of bits [Section 1.3.3.1 and 4.5]. No range specifications are possible for fixed or floating point data [Section 4.5].

Fixed point data is not considered exact, and so there is no provision for specifying a fixed point "step size" [Section 1.3.3.1].

No subscript range exception condition is supported by the language, although a compile time check for subscript range errors is performed when constant subscript values are used [Section 6].

D4. Numerical Range and Step Size (PASCAL)

Not Satisfied.

PASCAL does not have fixed point variables (see A2). The range of numeric variables - integer and real - plus the range of other scalar varables (e.g., character and enumeration) may be specified using subrange types. However, this is not required and "subrange" is regarded as a type in the language [p. 143].

D4. Numeric Range and Step Size (SIMULA 67)

Not Satisfied.

Individual ranges for numeric variables cannot be specified in SIMULA 67. Rather, a numeric variable can have either integer whole number range or real number range (which are "implementation defined") [B 72, B 375]. Fixed point variables, and hence their step sizes, are not available.

D4. Numeric Range and Step Size (TACPOL)

Partially Satisfied.

TACPOL only supports fixed point numerics [Section 3-2]. Their precision and scale must be declared [Section 4-1]; in effect, this gives a numeric variable's range, although ranges are limited to powers of two, as are step sizes.

D5. Variable Types (TINMAN)

The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type, or a contiguous subsequence of any enumeration type.

This permits arrays to be components of records or arrays and permits records to be components of arrays.

D5. Variable Types (ALGOL 68)

Partially Satisfied.

Algol 68 allows variables, array elements, structure fields (record components), procedure parameters and results, and pointed-to values to be of any type [Section 1.2.1, 4.4, 4.6]. However, the language does not have enumeration types.

D5. Variable Types (J3B)

Partially Satisfied.

A variable, array, or record component can be defined to be an built-in data type [Sections 4.2, 4.3, 4.4, 4.5, and 4.8]. Tables (i.e., arrays of records) are, however, limited to a single dimension, whereas arrays can have up to three dimensions, so arrays of records are not supported in the full generality implied by D5. In addition, arrays are not permitted as components of records (although parallel tables gives limited support to this concept).

D5. Variable Types (PASCAL)

Satisfied.

The component type of an array [p. 143], the type of a record section [p. 144], and the type of a variable [p. 146] can be any type in the language. This includes the built-in, the programmer defined, and the subrange types. For the implemented version PASCAL 6000 "it is not possible to construct a file of files; however, records and arrays with files as components are allowed" [p. 97].

D5. Variable Types (SIMULA 67)

Partially Satisfied.

Object attributes (i.e., record components) can be any built-in or defined type (or class) [CBL 2.2]. Array components are either value type (i.e., built-in) or reference type (i.e., pointer to defined type) [CBL 3.1], i.e., arrays of records are not supported. As remarked under A5 and B11, SIMULA 67 does not support enumeration types.

D5. Variable Types (TACPOL)

Partially Satisfied.

Arrays are restricted to components of scalar type [Section 4-2]. Groups may contain only elementary components or arrays [Section 4-5.a]. Tables are just one dimensional arrays of groups [Section 4-6]. Thus, TACPOL's array and

record features are severely restricted as to the structure of their components.

D6. Pointer Variables (TINMAN)

The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

Assignment to a pointer variable will mean that the variable's name is to act as an additional label (or reference) on the datum being assigned. Assignment to a nonpointer variable will mean that the variable's name it to label a copy of the object being assigned. For data without alterable component structure or alterable component values, there is no functional difference between reference to multiple copies and multiple references to a single copy. Consequently, pointer/nonpointer will be a property only of variables for composite types and of composite array and record components. Because the pointer/nonpointer property applies to all variables of a given type, it will be specified as part of the type definition. The use of pointers will be kept safe by prohibiting pointers to data structures whose allocation scope is narrower than that of the pointer variable.

Such a restriction is easily enforced at compile time using hierarchical scope rules providing there is no way to dynamically create new instances of the data type. In the latter case, the dynamically created data can be allocated with full safety using a (user or library defined) space pool which is either local (i.e., own) or global to the type definition. If variables of a type do not have the pointer property then dynamic storage allocation would be required for assignment unless their size is constant and known at the time of variable allocation. Thus, the nonpointer property will be permitted only for types (a) whose data have a structure and size which is constant in the type definition or (b) which manage the storage for their data as part of the type definition. Because pointers are often less expensive at run time than nonpointers and are subject to fewer restrictions, the specification of the nonpointer property will be explicit in programs (this is similar to the Algol-60 issue concerning the explicit specification of "value" (i.e., nonpointer) and "name" (i.e. pointer). The need for pointers is obvious in building data structures with shared or recursive substructures; such as, directed graphs, stacks, queues, and list structures. Providing pointers as absolute address types, however, produces gaps in the type checking and scope mechanisms. Type and access restricted pointers will provide the power of general pointers without their undesirable characteristics.

D6. Pointer Variables (ALGOL 68)

Satisfied.

Algol 68 provides pointer types, variables, array and record components [Section 1.2.1, 4.4, 4.6]. It permits definition of recursive, sharable, and dynamically alterable data structures (and substructures). The type of a pointer includes the type of the value pointed to, so pointers are compile time type restricted. The language prohibits assigning pointers to variables whose lifetime is longer than the object pointed to [Section 5.2.1].

D6. Pointer Variables (J3B)

Partially Satisfied.

J3B provides fully type-checked pointers. No data value can be accessed through a pointer without qualifying the pointer value with a data type, either in the pointer's declaration or at its point of use [Section 6]. J3B pointers can only be associated with records and arrays of records, i.e., tables, (in which case the pointer values can be associated with each record of the array) [Section 1.3.3.1, p. 1-16]. Although J3B supports pointer arithmetic and pointer comparisons [Sections 7.3 and 9.7] these operations are type-constrained to guard against error. The language does not, however, support any restriction against assigning a pointer value to a variable whose storage allocation lifetime may be longer than that of the object pointed to.

D6. Pointer Variables (PASCAL)

Not Satisfied.

PASCAL has a pointer type for building shared and recursive data structures [p. 145]. Assignment to a pointer variable is very restrictive in that only a pointer of the exact type may be assigned [p. 152]. The standard procedures new and dispose for explicit dynamic allocation mean pointer variables are not as safe in their use as are any other variables, since it is possible to deallocate record storage that is referenced by a still accessible variable.

D6. Pointer Variables (SIMULA 67)

Satisfied.

There is no implicit deferencing in SIMULA 67. In general, "the type of the value or reference obtained by evaluating the right part, must coincide with the type of the left part" (CB1 6.1.2). However, SIMULA 67 does have an extremely safe and flexible pointer mechanism because "an object-reference-assignment is always checked for legality (mainly at compile time)" [B 152]. This guarantees that a pointer is either NONE or references an object of the specified class.

D6. Pointer Variables (TACPOL)

Not Satisfied.

TACPOL does not have a pointer mechanism.

E1. User Definitions Possible (TINMAN)

The user of the language will be able to define new data types and operations within programs.

Definitions will have the appearance and costs of features which are built into the language while actually being catalogued accessible application packages. The operation definition facility will include the ability to define new infix operators (but see H2 for restrictions).

E1. User Definitions Possible (ALGOL 68)

Satisfied.

Algol 68 allows the programmer to define new data types [Section 4.2], new operators [Section 4.3, 4.5], and new procedures [Section 5.4.1].

E1. User Definitions Possible (J3B)

Not Satisfied.

Although record data can be defined as a data type, J3B provides no capability for defining new infix and prefix operators, nor for extending existing operations to new data types (as required by E4). In addition, programmer-defined functions may not return arrays or records [Section 3.3.3].

E1. User Definition Possible (PASCAL)

Partially Satisfied.

PASCAL provides an elegant "mechanism, type definition, for creating new types" [p. 18]. However, the only operations definition facility is the procedure or function. In PASCAL there is no ability to define new infix operators nor may programmer-defined functions return arrays or records.

E1. User Definitions Possible (SIMULA 67)

Partially Satisfied.

The SIMULA 67 class concept permits user definition of new types together with operations upon the data [CBL 2]. In a strict sense there is no ability

to define new infix operators. But one can write expressions of the form RAND1.OPERATOR(RAND2) where OPERATOR is a procedure defined with the class containing RAND1 as an object and applicable to objects of the class to which RAND2 belongs [B 128-134]. Effectively this is an infix operator definition capability.

E1. User Definitions Possible (TACPOL)

Not Satisfied.

TACPOL has no type or operator definition facility.

E2. Consistent Use of Types (TINMAN)

The "use" of defined types will be indistinguishable from built-in types.

Whether a type is built-in or defined within the base will not be determinable from its syntactic and semantic properties. There will be no ad hoc special cases nor inconsistent rules to interfere with and complicate learning, using, and implementing the language. Type definitions should be processed entirely at compile time and the language should allow full program specification of the internal representation.

E2. Consistent Use of Types (ALGOL 68)

Satisfied.

Algol 68 does not distinguish the use of newly defined types from built-in types. This is part of its orthogonal design [Section 0.1.2].

E2. Consistent Use of Types (J3B)

Not Satisfied.

User-defined operator extensions are not supported by J3B. In addition, arrays of records defined as a new J3B data type can only be accessed via pointer variables; normal subscripting is not allowed [Sections 4.2, 4.3, 4.4, and 6], nor can these types be returned as values of functions. Furthermore, matching rules are different for parameters declared with user-defined type names [Section 5.3.3].

E2. Consistent Use of Types (PASCAL)

Not Satisfied.

Type definitions are processed at compile time but PASCAL, by deliberate design, does not allow full program specification of the internal representation. Also, the language distinguishes between structured and non-structured types (whether built-in or user defined) in that a function's result type "must be a scalar, subrange, or a pointer type" [p. 162]. Thus, none of the structured types - array, record, set, or file - can be returned as a function value.

E2. Consistent Use of Types (SIMULA 67)

Partially Satisfied.

In fact, SIMULA 67 is extended from a common base language to a simulation language by providing the system classes (i.e., built-in types) SIMSET and SIMULATION exactly as user defined types available for prefixing. However, in contrast to user defined classes, the classes SIMSET and SIMULATION are distinct in that "an implementation may restrict the number of block levels at which such prefixes or block prefixes may occur in any one program" [CBL 14]. This is a small but real distinction between two built-in types and user defined types. Secondly I/O facilities are defined using a class called BASICIO, but this "class is not explicitly available in any user's program" [CBL 11]. User defined types cannot be hidden in this manner.

E2. Consistent Use of Types (TACPOL)

Not Satisfied.

As noted in E1, the user cannot define new types.

E3. No Default Declarations (TINMAN)

Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

This is a special case of requirement I1.

E3. No Default Declarations (ALGOL 68)

Satisfied.

Algol 68 has no default declarations. The two level syntax of the Revised Report carries the list of defined identifiers, modes, operators, etc. and thus allows only the defined ones to be applied (used) [Section 4.8, 7.2].

E3. No Default Declarations (J3B)

Satisfied.

All variables, including loop control variables, must be explicitly declared. There are no rules for implicitly defining undeclared identifiers [Section 4].

E3. No Default Declarations (PASCAL)

Satisfied.

"Every variable occurring in a statement must be introduced by a variable declaration" [p. 137]. Some types, procedures, and functions may be predefined (e.g., the standard input and output files - p. 164).

E3. No Default Declarations (SIMULA 67)

Partially Satisfied.

Explicit declarations are required for all program components except, for arrays where "if the type is omitted, the array is assumed REAL by default" [B 371].

E3. No Default Declarations (TACPOL)

Partially Satisfied.

"All names must be declared" [Section 10-1]. Specifically, names are declared in data or procedure declarations. Labels are declared by appearance and "file declarations may not appear outside the Compool" [Section 14-1]. Examination of actual TACPOL shows that loop control variables are implicitly declared to be FIXED (15), and are local to the loop in which they are used.

E4. Can Extend Existing Operators (TINMAN)

The user will be able, within the source language, to extend existing operators to new data types.

The translator will not assume that commutativity of built-in operations is preserved by extensions, and any assumptions about the associativity of built-in or extended operations will be ignored by the translator when explicit parentheses are provided in an expression.

E4. Can Extend Existing Operators (ALGOL 68)

Partially Satisfied.

In Algol 68, the programmer can extend existing operators to new types [Section 4.5]. However, a new type is represented in terms of the base types and the base type properties are inherited by the new type [Section 2.1.1.2, 7.3]. Algol 68 does not have generic procedures although it does support generic user defined operators.

E4. Can Extend Existing Operators (J3B)

Not Satisfied.

User-defined operator extensions are not supported by J3B.

E4. Can Extend Existing Operators (PASCAL)

Not Satisfied.

The permissible operand types for existing operators are completely circumscribed and are not extendable to new data types [pp. 149-151].

E4. Can Extend Existing Operators (SIMULA 67)

Not Satisfied.

Existing operator symbols cannot have their domain of operands extended. For example, "The constituents of simple arithmetic expressions must be of types real or integer" [AR 3.3.4].

E4. Can Extend Existing Operators (TACPOL)

Not Satisfied.

The TACPOL user cannot extend the domain of an existing operator symbol.

E5. Type Definitions (TINMAN)

Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

Types define abstract data objects with special properties. The definable operations will include constructors, selectors, predicates, and type conversions.

E5. Type Definitions (ALGOL 68)

Not Satisfied.

Algol 68 defined types inherit the properties and applicable operators of the component base types [Section 2.1.1.2, 7.3]. For example, if

mode matrix = [1:m, 1:m] real

and if multiplication '*' is already defined to be component-wise for two-dimensional arrays of real, matrix inherits the component-wise definitions of '*'. The syntax does not require a type's associated operator definitions to be lexically grouped with the type definition.

E5. Type Definitions (J3B)

Partially Satisfied.

J3B does not enforce the clustering of data type definitions with corresponding operations on these data types, although since a compilation unit can consist of several function definitions, user-defined operations on a user-defined data type can be grouped together [Section 3.1].

The rules for assignment and parameter passing of user-defined types require matching type names (rather than type representations) [Section 5.3.3], so different types having the same representation cannot be treated interchangeably by a set of operators. On the other hand, if a record type, D, is defined by adding new fields to a record type C, functions accepting objects of type C as arguments will also accept objects of type D. In this sense, a defined type inherits operations of the data with which it is represented. This kind of "inheritance" is also supported by SIMULA 67.

E5. Type Definitions (PASCAL)

Partially Satisfied.

A type definition in PASCAL has the form

<identifiers> = <type>

where a type is either a simple, a structured, or a pointer type [p. 111]. The set of applicable operators cannot be defined as a part of the type definition for any such types. Of course, procedure and functions may be separately specified to have parameters with newly defined types [p. 112].

A defined type does not inherit the operations of the data with which it is represented.

E5. Type Definitions (SIMULA 67)

Partially Satisfied.

SIMULA 67's class concept first introduced the definition of types as data objects *together with a set of applicable operations* [CBL 1.3, 2]. Selectors, predicates, and type converters can be programmed as procedures associated with a class. The object generator NEW serves as a constructor [CBL 4.3.2.2].

"A class declaration with the prefix 'C' and the class identifier 'D' defines a subclass D of the class C. An object belonging to the subclass ... is itself an object of the class C" [CBL 2.2.1]. This means operations defined in class C can also be applied to objects declared to be of class D, i.e. objects of a defined type D do, in this sense inherit operations of the type (C) in terms of which the definition is given. This does not actually conflict with the intent of E5, which takes a "bottom up" view of type definition, while SIMULA takes a "top down" view. It does however violate the letter of E5.

E5. Type Definitions (TACPOL)

Not Satisfied.

TACPOL does not have a type definition capability.

E6. Data Defining Mechanisms (TINMAN)

The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

E6. Data Defining Mechanisms (ALGOL 68)

Partially Satisfied.

Algol 68 has array and record (structure) types, along with fully discriminated union types [Section 1.2.1, 4.6]. However, it has neither enumeration types nor power set types. The language allows full compile time type checking [Section 0.1.4.1], but does not require it. Algol 68 provides both fixed and dynamic storage allocation [Section 5.2.3].

E6. Data Defining Mechanisms (J3B)

Partially Satisfied.

J3B supports a form of type definition limited to records and one dimensional arrays of records.

J3B does not support enumeration types, power sets, or array and record classes in their full generality (arrays and records cannot themselves be defined as array and record components).

Garbage collection and dynamic storage allocation are not supported by J3B.

E6. Data Defining Mechanisms (PASCAL)

Satisfied.

PASCAL permits defining types by enumeration, as arrays, records, discriminated unions (a record type with variants and a tag field), or sets [pp. 142-144]. Garbage collection, or at least dynamic storage allocation, is needed to support the new and the dispose standard procedures [p. 161].

E6. Data Defining Mechanisms (SIMULA 67)

Partially Satisfied.

SIMULA 67 supports neither enumeration types nor their power sets. A garbage collector is a standard, though optional [CBL 9.1], part of the runtime system [Structure, p. 6]. Arrays of records (i.e., classes) cannot be defined, although arrays of refs to classes can be.

Union (i.e., classes with prefixes) are fully discriminated [CBL 2.2.1], with necessary run time checks initiated through use of the INSPECT statement [B 150].

E6. Data Defining Mechanism (TACPOL)

Not Satisfied.

TACPOL does not support enumeration types; limited forms of arrays and record types are provided (see A7, D5). Cells, which "are used to overlay areas of storage" [Section 4-7], give a rudimentary free union facility. No discriminated union facility is provided. The bit string data type [Section 3-3.b] is TACPOL's substitute for the power set of an enumeration type.

E7. No Free Union or Subset Types (TINMAN)

Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.

Range and subset specifications on variables are useful documentation and debugging aids, but will not be construed as types. Subsets do not introduce new properties or operations not available to the superset and often do not form a closed system under the superset operations.

E7. No Free Union or Subset Types (ALGOL 68)

Satisfied.

Algol 68 unions are fully discriminated [Section 1.2.1, 2.1.1.2, 3.4, 4.6]. The language does not have subranges or subsets.

E7. Free Union or Subset Types (J3B)

Not Satisfied.

J3B permits the storage for one variable to overlay the storage occupied by another variable [Section 4.6]. Moreover, discrimination among record variants cannot be specified in the language, i.e., there are no compile time checks to enforce explicit type discrimination of records with variant structures.

E7. No Free Union or Subset Types (PASCAL)

Not Satisfied.

As indicated in A7, the tagfield of a variant record is optional [p. 144]. "If it is omitted, we obtain the equivalent of a free type union, and a compiler has no chance of checking consistency in its application" [Wirth, p. 197]. Range and subset specifications apppear in PASCAL as subrange types [p. 143]. Indeed, a run-time test to check "all assignments to variables of subrange types to make certain that the assigned value lies within the specified range" [p. 101] is an option for PASCAL 6000.

E7. No Free Union or Subset Types (SIMULA 67)

Satisfied.

Type unions are effected in SIMULA 67 by class prefixing [CBL 2.2.1]. Such type unions can be discriminated using the INSPECT statement [B 150].

E7. No Free Union or Subset Types (TACPOL)

Not Satisfied.

As noted in E6, CELLS [Section 4-7] provide a form of free union.

E8. Type Initialization (TINMAN)

When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

It is often necessary to do bookkeeping or to take other special action when variables of a given type are allocated or deallocated. The language will not limit the class of definable types by withholding the ability to define those actions. Initialization might take place once when the type is allocated (i.e., in its allocation scope) and would be used to set up the procedures and initialize the variables which are local to the type definition. These operations will be definable in the encapsulation housing the rest of the type definition.

E8. Type Initialization (ALGOL 68)

Not Satisfied.

Algol 68 does not directly associate procedures with type declarations [Section 4.2] or with storage allocation [Section 4.4, 5.2.3]. One can write allocation procedures, but they must be explicitly called, e.g.

```
                                        heap [m, n] real;
mode matrix = [m, n] real;
ref matrix a = newmatrix  (3, 5);
```

E8. Type Initialization (J3B)

Not Satisfied.

J3B does not support the type definition concept implied by E8, nor does it provide a means of defining subroutines associated with the allocation and deallocation of variables.

E8. Type Initialization (PASCAL)

Not Satisfied.

Type definitions do not admit any procedural specifications [pp. 142-146]. The procedure statement new (p, t1, ....,tn) "can be used to

allocate a variable of the variant [record] with tag field values t1, ..., tn" [p. 161], but no initialization of components can acccompany this allocation.

E8. Type Initialization (SIMULA 67)

Partially Satisfied.

When an object is generated (i.e., a type instance is allocated) the actual parameters are evaluated and passed; then "control enters the object through its initial begin" [CBL 4.3.2.2]. Executing a detach statement completes the initialization actions [CBL 4.3.2.2, B 265]. Upon executing the final code of the CLASS-body and encountering the final end of an object, it becomes "terminated". It may still be accessible via references and consequently not deallocated [B 123-124, CBL 4.3.2.2]. So, in SIMULA 67 code is executed upon type allocation, but is not necessarily associated with deallocation.

E8. Type Initialization (TACPOL)

Not Satisfied.

For TACPOL's closest approximation to a type definition capbility, namely groups, tables, and cells [Section 4-4], the user cannot specify initialization and finalization procedures for the type, nor can he define procedures to be invoked when variables of a type are allocated and deallocated.

F1. Separate Allocation and Access (TINMAN)

The language will allow the user to distinguish between scope of allocation and scope of access.

The scope of allocation or lifetime of a program structure is that region of the program for which the object representation of the structure should be present. The allocation scope defines the program scope for which own variables of the structure must be maintained and identifies the time for initialization of the structure. The access scope defines the regions of the program in which the allocated structure is accessible to the program and will never be wider than the allocation scope. In some cases the user may desire that each use of a defined program structure be independent (i.e., the allocation and accessing scopes would be identical). In other cases, the various accessing scopes might share a common allocation of the structure.

F1. Separate Allocation and Access Allowed (ALGOL 68)

Partially Satisfied.

Algol 68 has two storage allocation classes [Section 5.2.3]: location (LIFO stack) and heap (global to whole program, dynamic allocation, garbage collected automatically). The access rules for identifiers, mode indicators, operators, etc. are the standard Algol 60 block structure rules [Section 4.8, 7.2]. Algol 68 prohibits the accesss scope from exceeding the allocation scope [Section 5.2.1], although the use of heap storage implies that a variable's allocation scope is not a function of a program's static block structure.

F1. Separate Allocation and Access Allowed (J3B)

Partially Satisfied.

J3B supports ALGOL 60 block structure [Section 3.2], although blocks cannot be smaller than subroutines. In addition, variables declared local to a subroutine will be allocated storage on each subroutine entry if the subroutine is defined as a reentrant subroutine [Section 3.3.2, p. 3-16]. Since nested subroutine definitions are not permitted [Sections 3.2.2, 3.3.2, and 5.1], the notion that a nested subroutine's storage might be allocated when a containing subroutine is invoked cannot be supported.

F1. Separate Allocation and Access (PASCAL)

Partially Satisfied.

Variables declared at the procedure or function level "are accessible by their identifier. They exist during the entire execution process of the procedure (scope) to which the variable is local" [p. 145]. Such variables "are not known outside their scope" [p. 159]. "No conflict arises from a declaration redefining the same identifier within a program" [p. 160]. So, declared variables are allocated on function/procedure entry and freed upon exit. They are accessible only from statements in the statement part and from any procedure or function in the declaration part which does not redeclare the variable [pp. 159-106]. That is, scope of access can be distinguished as a subset from scope of allocation for declared variables. This is a restricted form of ALGOL 60 block structure. However, "variables may also be generated dynamically, i.e. without any correlation to the structure of the program" [p. 145]." These variables are allocated by the standard procedure new. "Access is achieved via a so-called pointer value" [p. 145]. In conjunction with the procedure dispose, run-time checks (which are not specified in the defining documents) would be required to ensure that the access scope is never wider than the allocation scope.

F1. Separate Allocation and Access Allowed (SIMULA 67)

Partially Satisfied.

Using classes the user can distinguish between scope of allocation and scope of access. A class object is allocated by the generator new [CBL

4.3.2.2]. Attributes of the object can be accessed from within (i.e., during the execution of its code) or remotely via object expressions referencing the object [CBL 7]. That is, an object is accessible whenever a pointer to it is accessible. "It is a consequence of the language structure that an object, at the time of its deletion, cannot be referenced by any computable <object reference> expression" [CBL 9.1]. The separation of allocation and access is essential to SIMULA 67's coroutine capability [CBL 9.2]. The user does not have control over deallocation, since if pointers to a class object are in use external to it, then the object's lifetime is not associated with the class's executable code.

F1. Separate Allocation and Access Allowed (TACPOL)

Partially Satisfied.

TACPOL has adopted standard block structure for access scope (and automatic storage for allocation) [Section 7-1]. There are no variables declared to be STATIC except, presumably, those declared in COMPOOLS. There is no way to imply that a variable declared in a nested subroutine is to be allocated when a containing subroutine is to be invoked.

F2. Limiting Access Scope (TINMAN)

> The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

Limited access on the call side provides a high degree of safety and eliminates nonessential naming conflicts without limiting the degree of accessibility which can be built into programs. The alternative notion, that all declarations which are external to a program segment should have the same scope, is inconvenient and costly in creating large systems which are composed from many subsystems because it forces global access scopes and the attendant naming conflicts on subsystems not using the defined items.

F2. Limiting Access Scope (ALGOL 68)

Partially Satisfied.

Algol 68 does not allow access to a structure to be limited at the point of definition. Access to indicators declared in outer ranges (blocks) is limited at the point of use only by redeclaring the indicator in an inner range [Section 4.8, 7.2]. The language does allow new local names to be associated with global components, e.g.

```
struct (real re, im) X;
begin
        ref real rx = re of X;
        rx := 3.5;
end
```

F2. Limiting Access Scope (J3B)

Partially Satisfied.

Subroutine names can be specified to be accessible only from within a compilation unit, i.e., they can be declared internal rather than external [Sections 3.3.2 and 3.3.3]. Identifiers associated with variables can be declared external by declaring the identifiers in a block declaration [Sections 3.3.2 and 4.1]. Otherwise, such identifiers are local to either the compilation unit or subroutine in which they are declared.

The use of COMPOOLS permits declarations to be separated into groups made selectively available to requesting compilation units [Section 3.1], but J3B does not support a hierarchy of scopes external to compilation units, because COMPOOLs cannot be referenced from within a COMPOOL declaration file [Section 3.1.3].

To some extent, new local names can be associated with separately defined program components through appropriate use of J3B macro definitions [Section 2.2.6], but there is no language capability directly supporting the redefinition of names when reusable program components are integrated.

F2. Limiting Access Scope (PASCAL)

Not Satisfied.

Specifying access capabilities and limitations is not part of a type definition in PASCAL [pp. 142-146]. Similarly, limited access (e.g. read-only) cannot be specified with the use of a defined structure [pp. 148-152]. Separately defined program components (i.e., functions, data types, etc.) cannot be given new local names by the programmer to avoid naming conflicts.

F2. Limiting Access Scope (SIMULA 67)

Partially Satisfied.

The set of operations (i.e., procedures) applicable to a defined type (i.e., a class object) cannot be controlled. All declared class operations are equally accessible [CBL 7]. For example, with the INSPECT statement, code will be executed as if it "were written inside the body of the inspected object" [B 150]. That is, internal representations and operations cannot be hidden. For implementations permitting separate compilation, new local names

can be associated with the procedure or class declarations; naming conflicts can thus be automatically avoided [CBL 15].

F2. Limiting Access Scope (TACPOL)

Partially Satisfied.

The user cannot specify limited access to separately defined structures where they are used. New local names cannot be associated with separately defined program components.

TACPOL does however, provide the ability to limit access to files declared in the Compool. The authorization list and access mode specification AUTH "specifies the program(s) which are authorized to open a file and the access which they are allowed... The access to the file will be one of the following: INPUT, OUTPUT, UPDATE" [Section 14-3.i].

F3. Compile Time Scope Determination (TINMAN)

The scope of identifiers will be wholly determined at compile time.

Identifiers will be declared at the beginning of their scope and multiple use of variable names will not be allowed in the same scope. Except as otherwise explicitly specified in programs, access scopes will be lexically embedded with the most local definition applying when the same identifier appears at several levels.

F3. Compile Time Scope Determination (ALGOL 68)

Satisfied.

In Algol 68, the access scope of identifiers, type indicators, operators, etc. is completely determined by the static program text and hence at compile time [Section 3.2, 4.8, 7.2]. The most local (lexically) definition is used for an applied occurrence of an indicator [Section 4.8]. Multiple declarations of the same identifier in the same lexical level are prohibited [Section 4.8, 7.1]. However, Algol 68 allows identifiers to be declared after their first lexical applied occurrence. Thus, in

```
real X := Y;
real Y := 2.7;
```

the value of Y is not defined when the first line is executed.

F3. Compile Time Scope Determination (J3B)

Satisfied.

Static block structure scoping rules are used to establish lexically embedded scopes for identifiers [Section 3.2]. Different definitions of variable names are not allowed within the same scope ("A name has one and only one meaning in a given scope" [Section 2.2.1, p. 2-5]). In particular, components of different record types cannot have the same name. However, variables of the same table type will necessarily have identically named components. In J3B, the potential ambiguity implied here is resolved by permitting typed tables to be accessed only with pointer values, which inherently specify which table variable is being referenced. If index values were permitted, then a qualified name capability similar to PASCAL's would be required.

Although the scope of a macro definition is completely determined at compile time, macro identifiers can be given alternative definitions within the same block structure scope [Section 3.4]. In addition, since macro invocations appearing in a macro body are not processed until the macro body containing them is invoked [p. 3-25], these embedded macros use the definition active in the context of the macro call rather than the definition applying in the context of the definition. This means the definition of a macro is not necessarily determined by the lexical structure of a program at the point where the macro is declared.

F3. Compile Time Scope Determination (PASCAL)

Satisfied.

Identifiers are declared local to procedures and functions [pp. 159, 162]. Non-local accessible identifiers are determined by the static program text (i.e., at compile-time). "A procedure may reference any variable global to it, or it may choose to redefine the name" [p. 69].

F3. Compile Time Scope Determination (SIMULA 67)

Satisfied.

The class concept is an extension to block structure that carries over its notion of local identifiers [CBL 1.3.3, CBL 2.2, B 122-124]. Virtual quantities provide a means for binding a name in one scope to a definition in another scope at compile time [CBL 2.2.3].

F3. Compile Time Scope Determination (TACPOL)

Satisfied.

A TACPOL program is organized into blocks that "define the scope of names" [Section 7-1]. This scope is wholly determined at compile time.

F4. Libraries Available (TINMAN)

A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

There will be broad support for libraries common to users of well-recognized application areas. Application libraries will be developed as early as possible.

F4. Libraries Available (ALGOL 68)

Not Satisfied.

Algol 68 says nothing about libraries but allows an implementation to use pragmats (compiler directives) [Section 9.2] to access libraries.

F4. Libraries Available (J3B)

Not Satisfied.

Application-oriented language extensions cannot be made available in J3B, and so libraries of them cannot be made available.

F4. Libraries Available (PASCAL)

Not Satisfied.

PASCAL does not support the library concept as defined by the TINMAN.

F4. Libraries Available (SIMULA 67)

Partially Satisfied.

The defining documents do not require broadly supported libraries for a variety of application-oriented data and operations. However, the system classes SIMSET and SIMULATION [CBL 14] along with I/O classes [CBL 11] constitute a primitive library prototype. The effective equivalent of application libraries has arisen in several simulation areas. "A main characteristic of SIMULA 67 is that it is easily structured towards specialized problem areas, and hence will be used as a basis for Special Applications Languages" [CBL PREFACE].

F4. Libraries Available (TACPOL)

Not Satisfied.

The language reference manual makes no mention of application oriented libraries. TACPOL does, however, offer built-in procedures for standard numeric operations such as EXP, SQRT, REM, ABS, MAX, MIN, logarithm, and trigonometric functions [Section A-2, A-3].

F5. Library Contents (TINMAN)

Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

Whether a library should contain source or object code is a question of implementation efficiency and should not be specified in the definition of the source language, but the source language description will always be available. Library routines written in other languages will not be prohibited provided the foreign routine has object code compatible with the calling mechanisms used in the Common HOL and providing sufficient header information (e.g., parameter types, form, and number) is given with the routine in Common HOL form to permit the required compile time checks at the interface.

F5. Library Contents (ALGOL 68)

Not Satisfied.

Algol allows an implementation to use pragmats [Section 9.2] to access libraries and provide for code written in other languages. Algol 68 does not define the internal syntax and semantics of pragmats. Algol 68 also allows an implementation dependent library prelude [Section 10.1] that augments the standard prelude (which defines the standard types, operators, I/O routines, etc.).

F5. Library Contents (J3B)

Partially Satisfied.

Subroutine declarations, record type declarations, and other data declarations can be kept in J3B compools [Sections 3.1 and 4.1]. The subroutine declarations are checked at compile time against subroutine definitions and calls in compiled programs accessing the COMPOOL containing the declarations [Sections 3.3 and 5.3.3]. Compools cannot hold subroutine definitions, however, and may not hold routines whose bodies are written in other source languages [Section 3.1.3].

F5. Library Contents (PASCAL)

Not Satisfied.

PASCAL does not support the library concept as defined by the TINMAN.

F5. Library Contents (SIMULA 67)

Not Satisfied.

The system classes SIMSET and SIMULATION - along with the subclasses linkage, link, head, and process - must, in effect, be maintained in a compile time accessible library since they are not in the base language or in individual programs [CBL 14]. The SIMULA 67 definition does not require such libraries, nor does it accommodate routines whose bodies are written in other source languages. Defined procedures and class declarations (i.e., what is definable in SIMULA 67) can be separately compiled [CBL 15].

F5. Library Contents (TACPOL)

Partially Satisfied.

"Procedures, declarations, data declarations, and file declarations may be included in the Compool" [Section 14-1]. Whether routines written in other source languages can also be included is not addressed in the language reference manual.

F6. Libraries and Compools Indistinguishable (TINMAN)

Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

F6. Libraries and Compools Indistinguishable (ALGOL 68)

Not Satisfied.

Algol 68 does not support libraries or compools. See also requirements F4 and F5.

F6. Libraries and Compools Indistinguishable (J3B)

Not Satisfied.

Source code for inline subroutines (see J5) or other text to be copied into a compilation unit must be contained in files that are separate from compool files [Section 2.2.5]. This distinction between compool files and text files is supported by the language COMPOOL directive (used to reference compool files) [Section 3.1.2] and the COPY directive [Section 2.2.5] (used to insert text files into source programs in place of the COPY directive).

J3B compools are not hierarchically definable, i.e., a compool cannot reference a compool [Section 3.1.3], so compools cannot be associated with various levels of programming activity. The use of particular compools cannot be restricted to any particular program or subsystem of programs.

F6. Libraries and Compools Indistinguishable (PASCAL)

Not Satisfied.

PASCAL does not support a library or a compool concept.

F6. Libraries and Compools Indistinguishable (SIMULA 67)

Partially Satisfied.

SIMULA 67 does not distinguish (in terms of accessing methods) between separately compiled classes and classes represented in source text terms [CBL 15.2]. But classes cannot be associated with various levels of programming activity, nor is the operation of a library/compool specified as part of the language definition.

F6. Libraries and Compools Indistinguishable (TACPOL)

Partially Satisfied.

As noted in F5, TACPOL has a Compool capability. Anything definable in the language "may be included in the Compool" [Section 14-1]. However, as described in the language reference manual, there is just a single Compool per execution. Thus, it is not possible in TACPOL to associate different Compools "with any level of programming activity from systems through projects to individual programs."

F7. Standard Library Definitions for Machine-Dependent Interfaces (TINMAN)

The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

The idea is not to provide all the many special cases in the language, but to provide a few general cases which will cover the special cases.

There is currently little agreement on standard operating system, I/O, or file system interfaces. This does not preclude support of one or more forms for the near term. For the present the important thing is that one be chosen and made available as a standard supported library definition which the user can use with confidence.

F7. Standard Library Definitions for Machine-Dependent Interfaces (ALGOL 68)

Satisfied.

Algol 68 provides standard procedures [Section 10.3] for establishing the interface to data sets/devices and for device independent I/O (both formatted and unformatted).

F7. Standard Library Definitions for Machine-Dependent Interfaces (J3B)

Not Satisfied.

No I/O interfacing capabilities are explicitly provided by J3B.

F7. Standard Library Definitions for Machine-Dependent Interfaces (PASCAL)

Satisfied.

PASCAL's file data type with its associated buffer variable and get, put, reset, and rewrite procedures [p. 161] hides a sequential file allocated on some secondary storage media. In PASCAL 6000 "files that exist outside the program (i.e., before or after program execution) may be made available" if they are "declared as file variables in the main program" [p. 91]. However, "if PASCAL is to serve for system construction purposes, then the file facility might be dropped entirely, because the very purpose would be the description of possible file mechanisms in terms of more primitive concepts" [Wirth, p. 195]. Machine independent interfaces to the machine dependent character set ordering [p. 15] and the maximum integer value [p. 13] are provided. File handling more elaborate than for sequential files and special hardware are not accommodated.

F7. Standard Library Definitions for Machine Dependent Interfaces (SIMULA 67)

Satisfied.

Four types of files - input, output, direct I/O, and printfile - have predefined machine independent interfaces [CBL 11.1.2] but "an implementation may restrict in any way the use of these classes for prefixing" [CBL 11.1.2]. No mention is made of special hardware interfaces, but they would certainly be handled using classes, as with standard input and output devices [B 190-205].

F7. Standard Library Definitions for Machine Dependent Interfaces (TACPOL)

Satisfied.

TACPOL supports open, close, read, write, and other natural operations for both serial and direct access files [Section 13-4, 13-5, 13-6, Table 13-1]. The files may optionally be declared partitioned, blocked, or buffered [Section 14-3]. Most important, the "media" field in a file declaration "specifies the type of device that the file will be allocated to" [Section 14-3.g]. Thus, TACPOL's file declaration and associated operations provide a reasonable standard for "machine independent interfaces" to machine dependent peripheral devices.

G1. Kinds of Control Structures (TINMAN)

The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling.

These mechanisms, hopefully, provide a spanning set of control structures. The most appropriate operations in several of these areas is an open question. For the present, the choice will be a spanning set of composable control primitives each of which is easily mapped onto object machines and which does not impose run time charges when it is not used. Whether parallel processing is real (i.e., by multiprocessing) or is synthesized on a single sequential processor, is determined by the object machine, but if programs are written as if there is true parallel processing (with no assumption about the relative speeds of the processors) then the same results will be obtained independent of the object environment.

It is desirable that the number of primitive control structures in the language be minimized, not by reducing the power of the language, but by selecting a small set of composable primitives which can be used to easily build other desired control mechanisms within programs. This means that the capabilities of control mechanisms must be separable so that the user need not pay either program clarity or implementation costs for undesired specialized capabilities. By these criteria, the Algol-60 "FOR" would be undesirable because it imposes the use of a loop control variable, requires that there be a single terminal condition and that the condition be tested before each iteration. Consequently, "FOR" cannot be composed to build other useful iterative control structures (e.g., FORTRAN "DO"). The ability to compose control structures does not imply an ability to define new control operations, and such an ability is in conflict with the limited parameter passing mechanisms of C7.

G1. Kinds of Control Structures (ALGOL 68)

Partially Satisfied.

Algol 68 has sequential [Section 3.2], conditional [Section 3.4], and iterative [Section 3.5] control structures. It allows recursive procedures. It has a parallel processing control structure [Section 3.3] and synchronization operators (up and down on integer semaphores) [Section 10.2.4]. However, the language does not have an exception handling mechanism. (Certain I/O conditions such as end of file are handled by providing procedures to be called when the condition occurs). Nor does it provide for asynchronous interrupts.

G1. Kinds of Control Structures (J3B)

Partially Satisfied.

Sequential, conditional, iterative, and recursive control structures are supported by J3B [Sections 5.1, 5.5, 5.6, and 3.3.2]. No support is provided, however, for parallel processing, exception handling, and asynchronous interrupt handling.

G1. Kinds of Control Structures (PASCAL)

Partially Satisfied.

Sequential, conditional, iterative, and recursive control structures are provided [pp. 153-157,159, 162]. Asynchronous interrupt handling, (pseudo) parallel processing, and exception handling are not present in standard PASCAL except, of course, for end of file and end of text line predicates [p. 163].

G1. Kinds of Control Structure (SIMULA 67)

Satisfied.

SIMULA 67's FOR-statement is the same as ALGOL 60's [AR 4.6] extended "to facilitate the processing of list structures" [CBL 6.2.2]. The iteration mechanisms, FOR and WHILE, only test at the beginning of loops [B 80-81, B 92]. The language has no control structures for exception handling and asynchronous interrupt handling. SIMULA 67 does have a co-routine or quasi-parallel sequencing capability [CBL 9.2]. But programs cannot be written as if there is true parallel processing since explicit "detach" and "resume" statements are required to specify the precise sequencing [CBL 9.2.1, 9.2.2].

G1. Kinds of Control Structures (TACPOL)

Partially Satisfied.

TACPOL has structured control mechanisms for sequential [Section the BEGIN block, Section 7-2], conditional [Section the IF statement, Section 8-1], and iterative control [Section the DO statement especially with the

WHILE option, Section 8-3]. One cannot easily build other desired sequential control forms without using the GOTO statement. Parallel processing is available in limited form for input/output where "by specification of a RETURN attribute for certain operations, the transmisssion occurs concurrently" [Section 13-2]. For just the exception conditions zero divide (ZDIV) and fixed overflow (FOFL) the user can specify merely "whether or not a snap procedure is to be called" [Section 12-1]. TACPOL does not provide for asynchronous interrupt handling or recursive procedures.

G2. The GOTO (TINMAN)

The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.

The language should not, however, impose unnecessary costs for its presence. The "GO TO" will be limited to explicitly specified program labels at the same scope level. Switches, designational expressions, label variables, label parameters and numeric labels are not desired. Switches here refer to the unrestricted switches which are generalizations of the "GO TO" and not to case statements which are a general form for conditionals(see G3). This requirements should not be interpreted to conflict with the specialized form of control transfer provided by the exception handling control structure of G7.

G2. The GOTO (ALGOL 68)

Not Satisfied.

Algol 68 allows jumps from inner to outer lexical scopes [Section 5.4.4, 4.8, 7.2]. Jumps from outer to inner levels are prohibited (by the two-level syntax). Although label is not an Algol 68 type, procedure is. Thus label variables are obtained via procedure variables and label values via procedures whose bodies are goto statements.

G2. The GOTO (J3B)

Partially Satisfied.

Label definitions are limited in their scope to a subroutine body in J3B, i.e., it is impossible to jump into or out of a procedure body using a goto statement [Section 5.4.1]. Moreover, labels are not permitted as parameters [Section 5.3.3].

On the other hand, switches are supported by J3B [Section 5.4.3].

G2. The GOTO (PASCAL)

Not Satisfied.

PASCAL permits jumps out of procedures and functions. Each label must be explicitly declared in the procedure block where it marks a statement [p. 153]. Any statement in the block (including one within an inner procedure) may GO TO the label [p. 31]. Thus, scope levels can be crossed by a GO TO. "The effect of jumps from outside of a structured statement into that statement is not defined" [p. 32]. Technically, this means jumps into loops are not forbidden; their effect is left to the implementor. In addition, PASCAL's labels are numeric.

G2. The GOTO (SIMULA 67)

Not Satisfied.

The GOTO is not limited to explicitly specified program labels at the same scope level. Rather, GOTO's can leave a class scope completely [CBL 9.2.4]. SIMULA 67 also supports switches [CBL 8.1], designational expressions [CBL 4.1.1], and label parameters [CBL 8.2].

G2. The GOTO (TACPOL)

Not Satisfied.

"A GOTO statement may be used in any block to transfer control to a point within the block itself or to a point in any containing block" [Section 9-2.b]. Thus, GOTOs are not limited to the same scope level. Moreover, label parameters [Section 11-5] and switches [Section A-7] are a part of TACPOL.

G3. Conditional Control (TINMAN)

The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

The conditional control operations will be fully partitioned (e.g., an "ELSE" clause must follow each "IF THEN") so the choice is clear and explicit in each case. There will be some general form of conditional which allows an arbitrary computation to determine the selected situation (e.g., Zahn's device [Donald E. Knuth, "Structured Programming with go to Statements," ACM Computer Surveys,r Vol. 6, No. 4, December 1974] provides a good solution to the general problem). Special mechanisms are also needed for the more common cases of the Boolean expression (e.g., "IF THEN ELSE") and for value or type discrimination (e.g., "CASE" on one of a set of values or subtype of a union).

G3. Conditional Control (ALGOL 68)

Partially Satisfied.

Algol 68 permits selection of alternatives based on the value of a Boolean expression, on the value of an integer expression, and on the current type of a discriminated union value [Section 3.4]. However, the conditional control structures are not fully partitioned since the else or out clauses are optional (and default to else skip), nor are case statement alternatives labeled when selection is based on integer values. No equivalent of Zahn's device is supported.

G3. Conditional Control (J3B)

Not Satisfied.

J3B supports an IF-THEN-ELSE construct, but it does not require the presence of an ELSE clause [Section 5.5]. J3B does not directly provide a control structure for discriminating the subtype of a value from a union type, nor does it provide a case statement control structure. In addition, there is no equivalent of Zahn's device for exiting from loops or conditional statements.

G3. Conditional Control (PASCAL)

Partially Satisfied.

PASCAL permits if-then-else conditional statements [p. 154] but the else clause is not required. Similarly the case statement is not constrained to be fully partitioned [p. 155], although case constitutents must be explicitly labeled. If the case expression evaluates to a label not in the case label list, then "the effect is undefined" [p. 31]. Discriminated unions are realized in PASCAL by records with alternative structures, and the PASCAL case statement can be used to discriminate among these variants. Loop and conditional statement exiting is not supported.

G3. Conditional Control (SIMULA 67)

Not Satisfied.

SIMULA 67 permits if-then conditionals without requiring a matching else in accordance with ALGOL 60 [CBL 6., AR 4.5.1]. The INSPECT statement is a general case-like construct that discriminates on type [B 150-151]. It allows any object expression to determine the selection. An optional OTHERWISE clause can make it fully partitioned [CBL 7.2.1], but its use is not required. A case construct for other than type discrimination is not provided. No equivalent of Zahn's device is supported.

G3. Conditional Control (TACPOL)

Not Satisfied.

The IF statement [Section 8-1] is TACPOL's only conditional control structure and "the ELSE clause is sometimes omitted" [Section 8-1]. That is, the IF is not required to be fully partitioned. No case statement or variant of Zahn's device is supported.

G4. Iterative Control (TINMAN)

> The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run time execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

In its most general form, a programmed loop is executed repetitively until some computed predicate becomes true. There may be more than one terminating predicate, and they might appear anywhere in the loop. The most common case is termination after a fixed number of iterations and a specialized control structure should be provided for that purpose (e.g., FORTRAN "DO" or Algol-60 "FOR"). Loop control variables are by definition variables used to control the repetitive execution of a programmed loop and as such will be local to the loop body, but at loop termination it will be possible to pass their value (or any other computed value) out of the loop, conveniently and efficiently.

G4. Iterative Control (ALGOL 68)

Partially Satisfied.

The Algol 68 iterative control structure [Section 3.5] is

for i from e1 by e2 to e3 while b do s od

The termination condition is tested before each iteration instead of anywhere during an iteration. However, the other requirements are satisfied (control variable is local to the loop, entry is only at the head (labels defined in loops are inaccessible from outside the loop), for and while loops are special cases of the general loop structure).

G4. Iterative Control (J3B)

Not Satisfied.

Although J3B provides an iterative control structure [Section 5.6], it does not permit the termination condition to appear anywhere in the loop, it does not require the control variable to be local to the loop, and it does not provide special case notations for a fixed number of iterations. In addition, the value of the loop control variable is implementation defined on loop exit and is directly accessible from outside the loop body. J3B does, however, insure that loops are entered only at their head, and not by jumping into the body of the loop [Section 5.6].

G4. Iterative Control (PASCAL)

Not Satisfied.

PASCAL's repetitive statements - while, repeat, and for - do not permit the termination condition to appear anywhere in the loop. In addition, although G4 does not require it, the PASCAL for statement does not permit incrementing or decrementing the control variable by steps other than unity.

Entry should only be at the loop head since the repetitive statements are structured statements. But jumping into such statements is undefined (i.e., it can vary with implementations) rather than forbidden [p. 32].

G4. Iterative Control (SIMULA 67)

Not Satisfied.

The FOR-statement control variable is not local to the loop [B 92]; SIMULA 67's iterative control structures, the FOR and WHILE statements, terminate "only at the head of the loop" [B 80, B 92]. GOTOs are restricted to lead only "to a program point within an operating block instance" [CBL 9.2.4]. The CBL defining document does not forbid a GOTO to a labeled statement within a FOR or WHILE construct. And the AR simply makes it "undefined" [AR 4.6.6] (i.e., implementation dependent) rather than forbidden.

G4. Iterative Control (TACPOL)

Partially Satisfied.

TACPOL does not have an iterative control structure that permits the termination condition to appear anywhere in the loop. The DO statement's control variable is local to the DO loop [Section 8-3], although the Reference Manual does not state this explicitly. In addition, the loop control variable is implicitly declared to be an integer variable.

G5. Routines (TINMAN)

Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

G5. Routines (ALGOL 68)

Partially Satisfied.

Algol 68 has both recursive and non-recursive routines [Section 5.4.1]. However, it allows procedures to be defined within the bodies of recursive procedures.

G5. Routines (J3B)

Satisfied.

Both recursive and non-recursive subroutines are supported by J3B [Section 3.3.2]. (Recursive subroutines are those routines declared to be reentrant.) Because no J3B subroutine can be defined within the scope of another subroutine [Sections, 3.2, 3.3.3, and 5.1], it is not possible to define subroutines within the body of a recursive subroutine.

G5. Routines (PASCAL)

Partially Satisfied.

"The use of the procedure identifier in a procedure statement within its *declaration implies recursive* execution of the procedure" [p. 159]. Similar remarks apply to functions [p. 162]. As with any procedure it is possible to define a procedure within the body of a recursive procedure.

G5. Routines (SIMULA 67)

Partially Satisfied.

Recursive procedures are permitted ( 164-173) but it is possible to define procedures within the body of a recursive procedure since SIMULA 67 subsumes ALGOL 60 declarations [CBL 2.1, AR 5.4.1].

G5. Routines (TACPOL)

Not Determinable.

It is not clear whether or not TACPOL supports recursion; the Reference Manual does not address this issue. However, TACPOL is characterized as "a modified subset of the PL/I language" [Section 1-1] and RECURSIVE is not a reserved word [Section 2-6.g.(3), Table 2-1]. Since PL/I requires explicit declaration of the RECURSIVE attribute, it could be concluded that TACPOL does not permit recursive procedures. On the other hand, since local variables are allocated on procedure invocation it would appear that procedures could be called recursively, although it is uncertain whether nested recursive procedures would execute correctly in all cases.

G6. Parallel Processing (TINMAN)

The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

The parallel processing capability will minimally provide the ability to define and call parallel processes and the ability to gain exclusive use of system resources in the form of data structures, devices, and pseudo devices. This latter ability satisfies one of the two needs for synchronization of parallel processes. The other is required in conjunction with real time constraints (see G8).

The parallel processing capability will be defined as true parallel (as opposed to coroutine) primitives, but with the understanding that in most implementations the object computer will have fewer processors (usually one) than the number of parallel paths specified in a program. Interleaved execution in the implementation may be required.

The parallel processing features of the language should be selected to eliminate any unnecessary overhead associated with their use. The costs of parallel processes are primarily in run time storage management. In particular, it will not be possible to define a parallel routine within the body of a recursive routine and it will not be possible to define any routine, including parallel routines, within the body of those parallel routines which can have multiple simultaneous activations. If the language permits several simultaneous activations of a given parallel process then it might require the user to give a upper bound on the number which can exist simultaneously.

G6. Parallel Processing (ALGOL 68)

Partially Satisfied.

Algol 68 has a true parallel processing control structure [Section 3.3]. It provides for synchronization via up and down operators on integer semaphores [Section 10.2.4]. However, it allows recursive procedures and parallel clauses to be arbitrarily nested.

G6. Parallel Processing (J3B)

Not Satisfied.

J3B provides no process control primitives.

G6. Parallel Processing (PASCAL)

Not Satisfied.

Standard PASCAL has no (possibly pseudo) parallel processing capability.

G6. Parallel Processing (SIMULA 67)

Not Satisfied.

SIMULA 67 does not support parallel processing with the ability to create and to terminate a process along with critical section control primitives. It does, however, provide a co-routine or quasi-parallel sequencing capability with appropriate creation, detaching, and resuming primitives [CBL 9.2].

G6. Parallel Processing (TACPOL)

Not Satisfied.

As mentioned in G1, TACPOL's parallel processing capability is limited to READ and WRITE statements where "if the RETURN option is present it specifies concurrent operation" [Section 13-6.f]. Otherwise processes cannot be created and terminated; consequently mutual exclusion is not provided for.

G7. Exception Handling (TINMAN)

The exception handing control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

The user must be able to specify the action to be taken on any exception situation which might occur within his program. The exception handling mechanism will be parameterized so data can be passed to the recovery point. Exception situations might include arithmetic overflow, exhaustion of available space, hardware errors, any user defined exceptions, and any run time detected programming error.

The user will be able to write programs which can get out of an arbitrary nest of control and intercept it at any embedding level desired. The exception handling mechanism will permit the user to specify the action to be taken upon the occurrence of a designated exception within any given access scope of the program. The transfers of control will, at the users option, be either forward in the program (but never to a narrower scope of access or out of a procedure) or out of the current procedure through its dynamic (i.e., calling structure). The latter form requires an exception handling formal parameter class (see C7).

G7. Exception Handling (ALGOL 68)

Not Satisfied.

Algol 68 has no exception handling control structure. Since it has procedure data types, routines may be assigned to data structures and passed as parameters and hence may be used for some exception handling, e.g., end of file.

G7. Exception Handling (J3B)

Not Satisfied.

J3B provides no exception handling capability other than the ability to write into a procedure or function's output parameters before returning from a call.

G7. Exception Handling (PASCAL)

Not Satisfied.

PASCAL does not have an exception handling control structure. It provides only end of file and end of text line text predicates [p. 163] and the capability of having functions or procedures as parameters [p. 152]. These can, in some cases, be used for exception handling that transfers control and data. However, such "procedures and functions must have value parameters only" [p. 158]. PASCAL 6000 provides as a user option, run-time tests for division by zero, array or case indexing error, integer overflow, and violation of subrange types [pp. 101, 121]. But such exception conditions produce only a run-time error and an informative dump [pp. 102-103].

G7. Exception Handling (SIMULA 67)

Not Satisfied.

As remarked under G1, SIMULA 67 does not support an exception handling control structure. Function procedures that test for a condition can be declared with a class (i.e., type). For example, the INFILE class has a Boolean procedure LASTITEM usable each time before attempting to read from INFILE [B 193-201, CBL 11.2.1]. Run time errors occur on arithmetic exceptions such as overflow [B 70-71].

G7. Exception Handling (TACPOL)

Partially Satisfied.

TACPOL gives the user control with ON statements [Section 13-13] over endfile conditions, trying to open a nonexisting file partition, or making a

record key error. One can CHECK for zerodivide or fixed overflow and get a diagnostic snapshot [Section 12-1]. This is the extent of a TACPOL programmer's control over error or exception situations.

G8. Synchronization and Real Time (TINMAN)

There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

When parallel or pseudo parallel paths appear in a program it must be possible to specify their relative priorities and to synchronize their executions. Synchronization can be done either through exclusive access to data (see G6) or through delays terminated by designated situations occurring within the program. These situations should include the elapse of program specified time intervals, occurrence of hardware interrupts and those designated in the program. There will be no implicit evaluation of program determined situations. Time delays will be program specifiable for both real and simulated times.

G8. Synchronization and Real Time (ALGOL 68)

Not Satisfied.

Algol 68 has none of the required features.

G8. Synchronization and Real Time (J3B)

Not Satisfied.

J3B provides no language features for dealing with parallel control paths, real time clocks, or asynchronous hardware interrupts.

G8. Synchronization and Real Time (PASCAL)

Not Satisfied.

Standard PASCAL has no synchronization or mutual exclusion primitives. It also has no DELAY-like statement. With respect to source level access to real-time clocks PASCAL 6000, for example, provides a predefined procedure time (), which assigns the current time as a character string [p. 98], and a predefined parameterless function, clock, which yields elapsed job time in milliseconds as an integer [p. 99].

G8. Synchronization and Real Time (SIMULA 67)

Not Satisfied.

SIMULA 67 does not admit "access to real time clocks" and does not have exception handling facilities for "asynchronous hardware interrupts" or other computational interrupts such as zerodivide or overflow. However, with respect to simulated time a comprehensive set of coroutine scheduling procedures are provided [CBL 14.2.4, B 286-288]. Delays and relative priorities are easily specified. Jacob Palme in "Making SIMULA into a Programming Language for Real Time" (1974 SIMULA Users' Conference) claims "that a small and simple addition to SIMULA can make the language suitable for real time applications".

G8. Synchronization and Real Time (TACPOL)

Not Satisfied.

TACPOL does not provide for asynchronous hardware interrupts, access to real time clocks, specification of relative priorities, or specification of time delays. After executing a WAIT statement, "the continued execution of the program is delayed until all operations requested pertaining to the designated file have been completed" [Section 13-14].

H1. General Syntax Characteristics (TINMAN)

The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

H1. General Characteristics (ALGOL 68)

Satisfied.

Algol 68 is free format, uses semicolon as a statement separator, allows arbitrary length identifiers, is based on conventional forms, has a uniform grammar, avoids special case notations, does not allow abbreviations, and is syntactically unambiguous. It is harder to parse than some languages because it allows the first use of a new type or operator to lexically precede the declaration of the type of operator.

H1. General Characteristics of Syntax (J3B)

Partially Satisfied.

J3B is a free format language, with the semicolon acting as a statement terminator [Sections 4 and 5]. Key words may not be abbreviated.

There are at least eight violations of syntactic uniformity, however:

- The ENDs terminating a compound statement cannot be labeled, i.e., one cannot write AA: END; [Section 5.1].
- Although a null compound statement is permitted (signified by BEGIN END;), a null set of table item and block declarations is not permitted, even though it also would be signified by the keyword sequence, BEGIN END; [Sections 4.1, 4.2, and 4.3];
- The BIT, BYTE, and INTR functions are permitted to take only variables as arguments [Section 9.2], although semantically there is no reason to forbid expresssions or constants.
- The syntax for defining macros differs unnecesarily from the syntax for defining subroutines in that a semicolon does not delimit the formal parameter list [Section 3.4];
- Arrays may have up to three dimensions, but tables can have at most one dimension;
- Typed tables can only be accessed with pointer values; untyped tables can only be accessed with integer values; arrays and array elements cannot be accessed through pointers, only through index values;
- Block items are initialized separately from their declaration, unlike other variables [Section 4.1, 4.4, and 4.5];
- Typed tables can only be initialized in a block, but untyped tables declared global to a compilation unit (or in a block) can be initialized as part of their declaration [Section 4.3 and 4.1].

H1 is also not satisfied in that only the first eight characters of an identifier are used to distinguish two identifiers [Section 2.2.1]. Consequently, identifiers can be abbreviated. The use of single letters to specify built-in data types in declarations [Section 4.8] might also be considered to violate H1's requirements for readability.

H1. General Syntax Characteristics (PASCAL)

Partially Satisfied.

In PASCAL "the semicolon (;) is considered as a statement separator, not a statement terminator (as e.g., in PL/I)" [p. 7]. The language is free format and permits long mnemonic identifiers. "Standard PASCAL will always recognize the first 8 characters of an identifier as significant" [p. 9]. In effect, this means identifiers can be abbreviated. Syntactic simplicity was a guiding idea in the language's design. PASCAL has an easily parsed one-token lookahead deterministic grammar [Wirth, SPE, p. 320].

H1. General Characteristics of Syntax (SIMULA 67)

Satisfied.

SIMULA 67 is free format, following ALGOL 60, which it extends [B 10]. The semicolon is used as a statement separator [B 79, B 363, AR 4.1.1]. Mnemonic identifiers [AR 2.4.1] with no abbreviations are allowed. The language is easily parsed using "the well-known transition table technique" [Structure, p. 4].

H1. General Characteristics of Syntax (TACPOL)

Partially Satisfied.

TACPOL is free format with a semicolon serving as a statement terminater [Section 2-2] and mnemonic identifiers can be used. "However, if more than eight characters are used for an identifier, the compiler will use only the first five and the last three characters for the identifier" [Section 2-6.g]. In effect, this means identifiers can be abbreviated. The key word SUBSTR for the string operator can also be written (abbreviated ?) as $ [Section 6-1.c, Table 6-3].

H2. No Syntax Extensions (TINMAN)

The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms, or define new infix operator precedences.

This requirement does not conflict with E4 and does not preclude associating new meanings with existing infix operators.

H2. No Syntax Extensions (ALGOL 68)

Not Satisfied.

In Algol 68, one can redefine existing operator precedences.

H2. No Syntax Extensions (J3B)

Satisfied.

No syntax extensions are permitted.

H2. No Syntax Extensions (PASCAL)

Satisfied.

No source language syntactic extensions are permitted the PASCAL programmer.

H2. No Syntax Extensions (SIMULA 67)

Satisfied.

The user cannot modify the source language syntax.

H2. No Syntax Extensions (TACPOL)

Satisfied.

TACPOL has none of the extension capabilities that H2 forbids.

H3. Source Character Set (TINMAN)

> The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

The language definition will specify the translation from the publication language into the restricted character set.

H3. Source Character Set (ALGOL 68)

Satisfied.

The character set of Algol 68 is suitable for publication purposes [Section 9.3, 9.4]. The language defines alternate representations using characters in the 64-character ASCII subset for those symbols, operators, etc. which are outside the subset [Section 0.1.4.5, 9.3, 9.4, 10.2.3].

H3. Source Character Set (J3B)

Satisfied.

All J3B symbols can be composed using the 64 character ASCII subset [Section 2.1]. No special character set has been specified for publication purposes, however.

H3. Source Character Set (PASCAL)

Partially Satisfied.

A CDC ASCII 64-character set version of PASCAL exists [pp. 94-95]. And the ASCII subset is more than adequate for expressing the source language syntax. But the defining documents do not specify the translation from a publication language into a restricted character set. For example, there is no representation dictum for the non-ASCII up-arrow.

H3. Source Character Set (SIMULA 67)

Not Satisfied.

The defining document does not specify translation into the ASCII 64 character set although a mapping between the formal language description symbols and the equivalent in sample print-outs is given [B 11]. SIMULA 67 uses the non-ASCII turnstile symbol for NOT [B 360]. Also, non-ASCII lower case letters are permitted in SIMULA 67 [AR 2.1, B 360].

H3. Source Character Set (TACPOL)

Satisfied.

All TACPOL programs can be written using just the 64 character ASCII subset. In fact, only 50 of the 64 ASCII characters are needed [Section 2-3]. There are even TACPOL conventions for representing a colon (..) or a semicolon (.,) when they are not in the available character set [Section 2-4]. Wherever non-ASCII characters are permitted (e.g, vertical bar for logical inclusive OR and double vertical bar for string catenation), ASCII alternatives are given (e.g, OR and CAT [Section Tables 6-3 and 6-4]).

H4. Identifiers and Literals (TINMAN)

The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

Some possible break characters are the space(E. W. Dijkstra, coding examples in Chapter I, "Notes in Structured Programming," in Structured Programming by O.-J. Dahl, E. W. Dijkstra and C.A.R. Hoare, Academic Press, 1972; and Thomas A. Standish, "A Structured Program to Play Tic-Tac-Toe," notes for Information and Computer Science 3 course at Univ. of California-Irvine, October 1974) (i.e., any number of spaces or end-of-line), the underline, and the tilde. The space cannot be used if identifiers and user defined infix operators are lexically indistinguishable, but in such a case the formal grammar for the language would be ambiguous (see H1). The language should require separate quoting of each line of a long literal:

"This is a long"

"literal string".

H4. Identifiers and Literals (ALGOL 68)

Partially Satisfied.

The Algol 68 Revised Report defines the formation of literals (8), identifiers, tokens, symbols, operators, key words, etc. [Section 9]. The break character for identifiers and numeric literals is space. However, there is no break character for character string literals. They must be continued at the start of the next line since separate quoting of each line is not allowed. Although one can use the concatenation operator to express the same effect, e.g. "ABC" + "DEF", there is no guarantee that the concatenation will be performed at compile time.

H4. Identifiers and Literals (J3B)

Partially Satisfied.

Formation rules for identifiers and literals are specified in the language definition [Sections 2.2.1 and 2.2.3]. The period and dollar sign may be used as break characters within identifiers [Section 2.2.1]. However, no break character is provided for use with numeric literals [Section 2.2.3]. Although separate quoting of each line of a long character or numeric literal is not required, the character literal format makes special provision for detecting a missing closing quote bracket, namely, semicolons can only be included in character literals by writing ';;' [Section 2.2.3.5]. Writing a single semicolon is an error. Since all statements are terminated by semicolons, a missing closing quote bracket is detected no later than the end of the statement containing the character literal.

H4. Identifiers and Literals (PASCAL)

Partially Satisfied.

Formation rules for identifiers and for literals are explicitly given [pp. 140-141]. But there is no break character for use internal to identifiers and literals, nor is separate quoting of each line of long literals required. The readability of identifiers can be enhanced using upper and lower cases, e.g., RealTimeClock.

H4. Identifiers and Literals (SIMULA 67)

Partially Satisfied.

The formation rules for identifiers [AR 2.4.1] and for literals - character constants [CBL 4.2.1, B 174], Boolean or logical values [AR 2.2.2], numbers [AR 2.5.1], text constants [B 178-179] - are explicit. There is no break character for use internal to identifiers and literals. Separate quoting of each line of a long literal is not required.

H4. Identifiers and Literals (TACPOL)

Partially Satisfied.

Though indicating the formation rules for identifiers [Section 2-6.g] and literals [Section 3-5], TACPOL does not permit a break character for use internal to identifiers and literals. Separate quoting of each line of a long literal is not required.

H5. Lexical Units and Lines (TINMAN)

There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

H5. Lexical Units and Lines (ALGOL 68)

Not Satisfied.

Algol 68 defines where typographical display features (space, new line, new page) may occur [Section 9.4.d], which is almost anywhere. In particular lexical units may be continued across lines. Some lexical units such as string literals or operations (:=) cannot contain arbitrary display features. The end of line character can be included in string literals as a data character if the implementation extends the production for other-string-item [Section 8.1.4.1.d] to include it, as stated in the Report.

H5. Lexical Units and Lines (J3B)

Not Satisfied.

Lexical units are specifically defined in J3B to be continuable across input record boundaries [Section 2.1, p. 2-3].

The end-of-line character can be included in literal strings if this character is supported by the implementation-defined character set [Sections 2.2.3.5 and 2.1].

H5. Lexical Units and Lines (PASCAL)

Not Satisfied.

Standard PASCAL does not mention the end-of-line for source programs. Thus, lexical units could cross line boundaries.

H5. Lexical Unit and Lines (SIMULA 67)

Not Satisfied.

SIMULA 67 programs are expressed in free format form with no consideration for line boundaries. Lexical units are not restricted to single lines.

H5. Lexical Units and Lines (TACPOL')

Not Satisfied.

Lexical units can continue across lines. For example, lines may contain up to 80 characters [Section 2-2], but "an identifier can be from one to any number of characters in length" [Section 2-6.g].

H6. Key Words (TINMAN)

Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

By key words of the language are meant those symbols and strings which have special meaning in the syntax of programs. They introduce special syntactic forms such as are used for control structures and declarations, or they are used as infix operators, or as some form of parenthesis. Key words will be reserved, that is unusable as identifiers, to avoid confusion and ambiguity. Finally, there will be no place in a source language program in which a key word can be used in place of an identifier. That is, functional form operations and special data items built into the language or accessible as a standard extension will not be treated as key words but will be treated as any other identifier.

H6. Key Words (ALGOL 68)

Satisfied.

Algol 68 key words are reserved, informative, and are not usable as identifiers [Section 9]. The language has about ninety bold face key words (counting syntactic words like if, types like real , and operators like le). Algol 68 key words are concise, e.g. int for integer.

H6. Key Words (J3B)

Partially Satisfied.

There are 52 key words in J3B [Section 2.2.1]. They are reserved.

Of the 52 key words, ABS, BIT, BYTE, ENTRY, FIX, INTGR, INTR, LBOUND, NENT, POINT, SCALE, SHIFTL, SHIFTR, and UBOUND can be used in syntactic contexts where a programmer-defined identifier can be used [Section 9].

H6. Key Words (PASCAL)

Satisfied (?)

PASCAL has 35 reserved keywords [p. 9]. Is this "very few in number"? The keyword mod is used as an infix operator [p. 13].

Requirement E1 includes "the ability to define new infix operators." (TINMAN, p. 34). If these new operators can be identifiers, then PASCAL's infix mod would appear "where an identifier can be used" (TINMAN, p. 49).

H6. Key Words (SIMULA 67)

Satisfied.

SIMULA 67 has 50 reserved key words [B 359]. They are a reasonable set that does not impair syntax recognition. The key words TRUE, FALSE, NONE, and NOTEXT can appear as constants in appropriate expressions [AR 2.2.2., CBL 4.3.1, CBL 4.4.1].

H6. Key Words (TACPOL)

Not Determinable.

This is an area of inconsistency in the language reference manual. Table 2-1 gives a list of 97 "reserved words" including such words as: B, CELL, E, L, S, and MOVE. And the rules for identifier formation state that "no identifier may be identical to any of the reserved words shown in Table 2-1" [Section 2-6.g.(2)]. However, this restriction on identifier names is ignored in Chapter 10 where one is cautioned that "care should be taken not to inadvertently redeclare names of intrinsic procedures" [Section 10-2.a]. An example is given wherein MOVE is used as a label; it is observed that "MOVE would no longer be accessible as an intrinsic procedure in that block or in any blocks contained within that block" [Section 10-2.a]. This contradicts the injunction against using MOVE as an identifier in Chapter 2. Similarly an example in Chapter 13 uses CELL as an identifier [Section 13-6.f.(2)], again contradicting Chapter 2.

H7. Comment Conventions (TINMAN)

The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear,

will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

Comments anywhere reasonable in a program will not be taken to mean that they can appear internal to a lexical unit, such as, an identifier, key word, or between the opening and closing brackets of a character string. One comment convention which nearly meets these criteria is to have a special quote character which begins comments and with either the quote or an end-of-line ending each comment. This allows both embedded and line-oriented comments.

H7. Comment Convention (ALGOL 68)

Partially Satisfied.

Algol 68 comments begin/end with a cents symbol, #, co, or comment. The report defines where they may appear [Section 9.1, 9.2], which is almost anywhere, but not inside certain lexical units such as key words and string literals. Comments may contain any character except the opening comment symbol. Automatic reformatting is not prevented. The requirement not satisfied is that end of line does not terminate a comment. Algol 68 also has pragmats, delimited by pragmat or pr, which are comments that an implementation may interpret as directives to the compiler, e.g., pr copy file xxx pr.

H7. Comment Conventions (J3B)

Partially Satisfied.

Comments are enclosed in either quote (") or percent (%) characters [Section 2.2.4]. Since quote characters are also used to delimit macro bodies and (possibly) actual macro parameters [Section 2.2.6] comments delimited with quote characters can potentially be confused by readers with source code not serving as a comment.

Any combination of characters can appear in a comment except a semicolon [Section 2.2.4]. The position of comments in programs is not unreasonably restricted [Sections 2.2 and 2.2.6]. However, comments do not automatically terminate at the end of a line. (Note however that a missing end of comment symbol is detectable because semicolons, i.e., statement delimiters, are not permitted in comments.)

H7. Comment Conventions (PASCAL)

Partially Satisfied.

Comments in PASCAL are demarked by curly brackets, { and }. On systems where they are not available, "the character pairs (* and *) are used in their place" [p. 9].

A comment "may be inserted between any two identifiers, numbers, or special symbols" [p. 9]. But end-of-line in the program text does not automatically terminate a comment.

H7. Comment Conventions (SIMULA 67)

Not Satisfied.

SIMULA 67 has two, rather than one, comment convention [B 360]. Comments are introduced by either the COMMENT or the END keyword and are terminated by a semicolon [B 360]. After the END keyword certain character combinations are forbidden, specifically: END, ELSE, OTHERWISE, and WHEN [B 360]. Comments do not automatically terminate at end-of-line if not otherwise terminated.

H7. Comment Conventions (TACPOL)

Partially Satisfied.

"Comments are permitted wherever blanks are allowed or required in a program... The character pair /* indicates the beginning of a comment and the same characters reversed */ indicate its end" [Section 2-5]. However, comments do not automatically terminate at end-of-line.

H8. Unmatched Parentheses (TINMAN)

The language will not permit unmatched parentheses of any kind.

Some programming languages permit closing parentheses to be omitted. If, for example, a program contained more "BEGINs" than "ENDs" the translator might insert enough "ENDs" at the end of the program to make up the difference. The language will require full parentheses matching. This does not preclude syntactic features such as "case x of s1, s2, ..., sn end case" in which "end" is paired with a key word other than "begin." Nor does it alone prohibit open forms such as "if-then-else-."

H8. Unmatched Parentheses (ALGOL 68)

Satisfied.

Algol 68 requires every opening symbol (e.g., if, begin, (, case, do) to have a closing symbol (e.g. fi, end), esac, od). Multiple closure is not allowed.

H8. Unmatched Parentheses (J3B)

Satisfied.

Implicit closure of compound statements is not permitted in J3B.

H8. Unmatched Parentheses (PASCAL)

Satisfied.

PASCAL does not support multiple closures. This can be verified by direct inspection of the language's syntax charts [pp. 116-118].

H8. Unmatched Parenthesis (SIMULA 67)

Satisfied.

SIMULA 67 does not support multiple closure.

H8. Unmatched Parentheses (TACPOL)

Satisfied.

All parentheses are matched including an END paired with the "four types of blocks; BEGIN, DO, PROC, and CODE" [Section 7-1]. Interestingly TACPOL, through essentially a subset of PL/I, does not permit labeled ENDs and multiple closures.

H9. Uniform Referent Notation (TINMAN)

There will be a uniform referent notation.

There will be no language imposed syntactic distinction between function calls and data selection. This does not preclude the inclusion of more than one referent notation.

H9. Uniform Referent Notation (ALGOL 68)

Partially Satisfied.

Algol 68 uses parentheses for procedure argument lists [Section 5.4.2, 5.4.3] and square brackets or parentheses for array subscript lists [Section 5.3.2, 9.4.1.f]. However, record components (structure fields) are selected by writing [Section 5.3.1] field of structure. The proceduring coercion (see B8) permits a parameterless routine to be called when an empty parameter list is not applied. Hence a variable can be replaced with a function implementation.

H9. Uniform Referent Notation (J3B)

Partially Satisfied.

The extent to which a language supports the uniform referent concept can be determined by answering two general questions: 1) can references to variables be replaced with invocations of subroutines merely by changing the declaration of the identifier representing the variable; and 2) can composite data structures be physically (but not logically) reorganized in a declaration without having to change any references in a program to these data structures or their components (i.e., a data structure's physical organization should not constrain the notation used to access it). J3B partially satisfies these criteria. Although array (table) and subroutine references use a uniform syntactic notation, parameterless function calls must use an empty parameter list enclosed in parentheses [Section 5.3.2], meaning that scalar variable references cannot be replaced with function calls simply by changing the declarations of an identifier. In addition, the uniform referent concept is also violated by J3B's inability to invoke functions as the target of assignment statements; this also implies references to variables cannot uniformly be replaced with function invocations.

H9 is also violated in that although a J3B indexed table is semantically similar to a one dimensional array of records, the method of accessing table entries (i.e., a particular record in the array of records) is not syntactically equivalent to the method of accessing array elements, since the keyword ENTRY must be used when accessing tables [Sections 9.5 and 4.3, p. 4-9], and this keyword cannot be used to access array components. In addition, J3B array elements cannot be accessed with pointers [Section 6], so although a table with a single table item is logically similar to an array, different accessing restrictions are imposed depending on whether the name being referenced is declared as a table or as an array.

Any "serially" organized table [Section 4.3] cannot be organized as a "parallel" table, since multiple word items may not be included in parallel tables [Section 4.3]. Consequently, serial and parallel tables cannot be used interchangeably with full generality. However, for tables not containing multiple word items, the serial/parallel options support the uniform referent concept, since the notation used to access components of such tables does not have to be changed when the organization is changed from serial to parallel (or vice versa).

Similarly, since ordinary tables cannot be passed as parameters, ordinary and specified tables cannot be used interchangeably (even though the only essential difference between them is the degree of control a programmer has over the physical layout of the table). This also violates the uniform referent principle.

On the other hand, a uniform notation is used whether table or table components are accessed with pointer or index values.

H9. Uniform Referent Notation (PASCAL)

Not Satisfied.

Array references use square brackets to enclose the index expressions [p. 147] while function designators have their actual parameter list demarked by parentheses [p. 151]. On the other hand, since parameterless functions can be invoked without supplying an (empty) argument list, a variable can be replaced with a function call. However, an identifier associated with a field of a record cannot be implemented as a function. For example, a tag field's value cannot be computed by a user -defined function as would be possible if the notation Record.tag, could be interpreted as a function call, tag (Record).

The ability to physically reorganize data structures without affecting how they are referenced notationally is also not fully supported in PASCAL. For example, BOX(I).WEIGHT would be written to access an array of records declared as:

```
box:  array [1..100] of record
                        weight:  integer;
                        girth :  integer;
                        end
```

whereas BOX.WEIGHT(I) would have to be written if the structure were reorganized as:

```
box:  record
                        weight:  array [1..100] of integer;
                        girth :  array [1..100] of integer
```

This physical reorganization of elements is supported uniformly, in contrast, by various JOVIAL dialects, where the first organization is called a "serial" table and the second, a "parallel" table. In JOVIAL, WEIGHT(I) would suffice to access components of BOX, whether its organization were serial or parallel, thus supporting the uniform referent concept.

H9. Uniform Referent Notation (SIMULA 67)

Not Satisfied.

SIMULA 67 takes its syntax for arrays and procedures from ALGOL 60. That is, arrays are declared [AR 5.2.1] and referenced [AR 3.1.1] using square brackets. Procedures are declared and invoked using parentheses (AR 3.2.1). For representation using standard printers, parentheses may serve as array brackets but the belief is that "A(I, J) loses much of the clarity of the formal A[I, J]" [B 11].

H9. Uniform Referent Notation (TACPOL)

Satisfied.

Array [Section 4-2], group [Section 4-5], table [Section 4-6], and cell [Section 4-7] component references all have the same form as function procedure calls [Section 9-3, 11-1.) .

H10. Consistency of Meaning (TINMAN)

No language defined symbols appearing in the same context will have essentially different meanings.

In particular, this would exclude the use of = to imply both assignment and equality, would exclude conventions implying that parenthesized parameters have special semantics (as with PL/1 subroutines), and would exclude the use of an assignment operator for other than assignment (e.g., left hand side function call. It would not, however, require different operator symbols for integer, real or even matrix arithmetic since these are in fact special cases of the same abstract operations and would allow the use of generic functions applicable to several data types.

H10. Consistency of Meaning (ALGOL 68)

Partially Satisfied.

In general, Algol 68 is uniform and consistent in its use of symbols with the exception that = is used to represent both the equality operator and to specify the value being assigned an identifier in an identity declaration. Although left hand side procedure calls are allowed (if they return a ref value) [Section 5.2.1], this is entirely consistent with the notion of assignment as supported by Algol 68.

Some of the operators have different meanings, depending on the type of the operand [Section 10.2.3.5.g, 10.2.3.7.5, 10.2.3.8.g, 10.2.4.e]. For example, + denotes concatenation when applied to strings; abs applied to a character converts it to an integer value; the interpretation of up depends on whether it is applied to a bits variable (when it means, shift left) or a semaphore variable.

H10. Consistency of Meaning (J3B)

Not Satisfied.

The = symbol is used for both assignment and equality in J3B [Sections 5.2 and 7.3.].

Real and integer division are both represented by / [Section 7.1], although the results of integer and real division are sufficiently different that different symbols should be used, as in ALGOL.

The colon is used to separate upper and lower bound specifications in table declarations [Section 4.3], to declare labels [Section 5.4.2], and to separate input from output parameters [Section 3.3.2, 3.3.3, 5.3.1 and 5.3.2].

The quote symbol (") is used to delimit comments [Section 2.2.4], macro bodies [Section 3.4], and actual macro parameters [Section 2.2.6].

Assigning to the name of a function is the method used to specify the value to be returned by the function, i.e. a special interpretation of the assignment operation is imposed when the target of the assignment is the name of a function. (Note that this makes it impossible for a compiler to ensure that a function always returns a well defined value at run time [Section 5.2 and 5.3.2].) The letter S has different meanings depending on its context. For example, the first occurrence of S in the declaration below specifies that TT is a serial table; the second occurrence specifies that II is a signed integer variable:

```
TABLE TT (5) S 8;
BEGIN
   ITEM II S 15;
   ...
END;
```

One-origin indexing is used to specify the number of words per table entry [Section 4.3], but zero-origin indexing is used elsewhere in the language, including the syntax for table packing [Section 4.5].

H10. Consistency of Meaning (PASCAL)

Partially Satisfied.

PASCAL substantially satisfies this requirement, but it does have at least one notable form of syntax sharing, namely, X followed by up-arrow denotes a buffer variable if X is of type file and it denotes the variable referenced by X if X is a pointer [p. 148].

H10. Consistency of Meaning (SIMULA 67)

Not Satisfied.

SIMULA 67 supports a PL/I-like "pseudo-variable" facility. For example, if T is a TEXT variable, then T.SUB(7, 2) can be used to denote a substring receiving field or a substring value depending on whether it appears on the left or the right of an assignment [B 183]. Further, in contrast to user defined procedures, "systems defined procedures may not be transmitted as parameters" [CBL 16].

H10. Consistency of Meaning (TACPOL)

Not Satisfied.

TACPOL permits a "substring operation on the left of an assign statement" [Section 6-1.c.(2)(b)]. Thus, the SUBSTR operation delivers a receiving field or a value. Further, the = symbol can stand for the equal relational operator [Section Table 6-2] and also the assignment operator [Section 5-1]. The keyword INIT is used to indicate both that an identifier has a constant value and to define the value [Section 4-9].

I1. No Defaults in Program Logic (TINMAN)

There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

The only alternative is implementation dependent defaults with the translator determining the meaning of programs. What a program does, should be determinable from the program and the defining documentation for the programming language. This does not require that binding of all program properties be local to each use. Quite the contrary, it would, for example, allow automatic definition of assignment for all variables or global specification of precision. What it does require is that each decision be explicit: in the language definition, global to some scope, or local to each use. Omission of any selection which affects the program logic will be treated as an error by the translator.

I1. No Defaults in Program Logic (ALGOL 68)

Satisfied.

I1. No Defaults in Program Logic (J3B)

Not Satisfied.

The following situations are defined to yield undefined effects by the J3B language specification, and these situations are such that the effects may differ for different translators for the same object machine:

- access the value of a loop variable from outside a loop after the loop has been exited [Section 5.6];
- returning from a function without assigning a value to the function [Section 5.4.1];
- executing a switch statement with an out of range value [Section 5.4.1];
- calling a subroutine with a negative integer actual parameter when the corresponding formal parameter is declared to be unsigned [Section 5.3.3];

- comparing a pointer value with NULL in a relational formula using <, <=, >, >= or comparing two pointer formulas that both point outside the table for which they were constructed [Section 7.3];
- using a BIT, BYTE, or INTR function with a negative index value, or specifying an index value and length whose sum is greater than the length of the value being extracted [Section 9.2];
- specifying a negative shift [Section 9.6].

Furthermore, the precision of fixed and floating point data is implementation defined, as is the character code used for character strings.

I1. No Defaults in Program Logic (PASCAL)

Not Satisfied.

The first character of a textfile sent to a printer is used as a printer control character "in some installations" [p. 60]. If a case selector has a value that does not equal any of the case labels, the effect is undefined [p. 31]. "The final value of a [loop] control variable is left undefined upon normal exit from the FOR statement" [p. 24]. "The predecessor of a character ... is undefined if one does not exist" [p. 15]. The effect of jumping into a loop from outside the loop is undefined rather than forbidden [p. 32].

I1. No Defaults in Program Logic (SIMULA 67)

Not Satisfied.

"The effect of text assignment is implementation defined if the left part and right part of the assignment refer to overlapping texts" [CBL 10.6].

I1. No Defaults in Program Logic (TACPOL)

Not Satisfied.

On assignment, "low order bits lost due to conforming to the quantity allocation of an identifier are truncated" [Section 5-2.a]. Thus, assignment of 40960 (= 2**16 + 2**14) to a short numeric identifier gives an actual value of 8192 (= 2**14) even though "the value is said to be undefined" [Section 5-2.a.(2)].

The effect of executing a switch statement with an out of range value is not specified in the reference manual [Section A-7], nor is it specified whether the value of a loop control variable is defined on loop exit.

I2. Object Representation Specifications Optional (TINMAN)

Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does

not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

Defaults will be allowed, in fact, encouraged in don't care situations. Such defaults will include data representations (see J4), open vs. closed subroutine calls (see J5), and reentrant vs. nonreentrant code generation.

I2. Object Representation Specifications Optional (ALGOL 68)

Partially Satisfied.

Algol 68 does not define any specific way for the programmer to specify object representations or override the defaults. However, one can use pragmats [Section 9.2] as implementation dependent compiler directives.

I2. Object Representation Specifications Optional (J3B)

Satisfied.

The J3B "specified" table capability permits programmers to describe how the fields of a record are arranged within the space occupied by the record [Sections 4.3 and 4.5]. A different form of table declaration permits the arrangement of fields to be determined by the compiler, although the programmer may specify general directives defining how space-efficiently the fields are to be packed [Section 4.3]. In addition, the physical arrangement of fields in an array of records (a table) is normally "serial"; this arrangement can be changed to "parallel" without changing the logical properties of the table [Section 4.3].

Whether a subroutine is compiled as open is not determined by default; the programmer must specify [Section 3.3.4]. In addition, reentrant code generation must be specified explicitly [Sections 3.3.2 and 3.3.3].

The physical organization of arrays in storage is prescribed explicitly in the language specification [Section 4.4], but there is no capability for programmers to specify a different physical organization.

The physical ordering in storage of variables, arrays, and tables is normally determined in an implementation dependent manner. The OVERLAY statement, however, may be used to physically reorder the arrangement of these objects to insure, for example, that certain addressing optimizations may be performed.

I2. Object Representation Specifications Optional (PASCAL)

Partially Satisfied.

Structured types, such as arrays and records, can be optionally designated as packed. "This has no effect on the meaning of a program, but it is a hint to the compiler that storage should be economized een at the price of some loss in efficiency of access" [p. 143]. However, object representation specifications for open vs. closed subroutine calls or for reentrant code are not available to the PASCAL programmer. And "a component of a packed structure must not appear as an actual variable parameter" [p. 83]. Interestingly, PASCAL 6000 has an option (and default) for using registers to pass parameters which improves code quality but runs the risk of a "running out of registers" error message [p. 101].

I2. Object Representation Specifications Optional (SIMULA 67)

Not Satisfied.

The SIMULA 67 user cannot optionally override the default representation of class objects.

I2. Object Representation Specifications Optional (TACPOL)

Partially Satisfied.

With arrays "the coder may change the normal aligned allocation for arrays to a packed allocation by using the PACKED option in the array declaration" [Section 4-3]. Groups, tables, and cells are normally PACKED, but this can be changed "by using the ALIGNED option" [Section 4-5.6].

However, open subroutines cannot be specified.

I3. Compile Time Variables (TINMAN)

The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

When a language has different host and object machines and when its compilers can produce code for several configurations of a given machine, the programmer should be able to specify the intended object machine configuration. The user should have control over the compile time variables used in his program. Typically they would be associated with the object computer model, the memory size, special hardware options, the operating system if present, peripheral equipment or other aspects of the object machine configuration. Compile time variables will be set outside the program, but available for interrogation within the program (see I4 and C4).

I3. Compile Time Variable (ALGOL 68)

Partially Satisfied.

Algol 68 does provide a variety of environment enquiry capabilities [Section 10.2.1], e.g. max real is defined as the largest value of type real. Implementations may provide additional environment information in a library prelude [Section 10.1]. The specified set of environmental enquiries does not, however, include object computer model, memory size, etc.

I3. Compile Time Variables (J3B)

Not Satisfied.

There is no provision for environmental enquiries in J3B, although a variety of machine dependent parameters are specified in the language specification [Section 1.3.1].

I3. Compile Time Variables (PASCAL)

Not Satisfied.

PASCAL was designed and defined "without reference to any particular machine in order to facilitate the interchange of programs" [p. 168]. Nothing about the intended object machine configuration is programmer specifiable. An implementation standard identifier, maxint, can be used within a program [pp. 13-14] and is the only such environmental enquiry available in the language.

I3. Compile Time Variables (SIMULA 67)

Not Satisfied.

No provisions for environmental enquiries exist.

I3. Compile Time Variables (TACPOL)

Not Satisfied.

TACPOL as defined in the language reference manual has a single object computer, the AN/GYK-12 [Section 7-5]. So, the language does not have compile time variables.

I4. Conditional Compilation (TINMAN)

The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and

other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

An environmental inquiry capability permits the writing of common programs and procedures which are specialized at compile time by the translator as a function of the intended object machine configuration or of other compile time variables (see I3). This requirement is a special case of evaluation of constant expressions at compile time (see C4). It provides a general purpose capability for conditional compilation.

I4. Conditional Compilation (ALGOL 68)

Not Satisfied.

Algol 68 does not define any conditional compilation capability. An implementation could use pragmats [Section 9.2] to provide it.

I4. Conditional Compilation (J3B)

Partially Satisfied.

If the predicate in a conditional statement is composed of bit constants [Section 8.3], the predicate is evaluated at compile time and object code is only generated for the THEN branch of the statement or the ELSE branch, if one is present [Section 5.5]. However, any macro definitions contained in the THEN or ELSE branches are processed regardless of the value of the predicate, i.e., conditional compilation of macro definitions is not possible [Section 5.5].

It should be noted, however, that the generality of this form of conditional compilation is limited in J3B since relational expresssions are not evaluated at compile time (an oversight in the language definition) [Section 8.3] and, since declarations must appear before executable statements [Section 3.1.1], declarations cannot be conditionally compiled. When subroutines are compiled in-line, conditional compilation is possible if some of the subroutine's actual parameters are constant [Section 3.3.4].

The term "conditional statements" in the TINMAN statement of I4 presumably rules out the conditional non-compilation of loop bodies when the predicate controlling loop iteration is a constant evaluable at compile time. J3B does not support this form of conditional compilation, although there appears to be no reason why it should not be supported.

I4. Conditional Compilation (PASCAL)

Not Satisfied.

Conditional compilation is not a PASCAL capability.

I4. Conditional Compilation (SIMULA 67)

Not Satisfied.

SIMULA 67 contains no conditional compilation capability.

I4. Conditional Compilation (TACPOL)

Not Satisfied.

TACPOL does not support a conditional compilation capability.

I5. Simple Base Language (TINMAN)

The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capablity not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

Only the base need be implemented to make the full source language capability available.

Base features will provide relatively low level general purpose capabilities not yet specialized for particular applications. There will be no prohibition on a translator incorporating specialized optimizations for particular extensions. Any extension provided by a translator will, however, be definable within the base language using the built-in definition facilities. Thus, programs using the extension will be translatable by any compiler for the language but not necessarily with the same object efficiency.

I5. Simple Base Language (ALGOL 68)

Partially Satisfied.

One of the design goals of Algol 68 is orthogonality [Section 0.1.2], i.e. a small number of independent concepts that may be arbitrarily combined. The Revised Report defines a base language and a standard environment [10] that defines the specific types, operators, and procedures generally available. For example, the addition operator is defined by extension, although implementations will of course treat addition as a built-in operator, to improve "object efficiency." Many of these procedures can be implemented using a subset of Algol 68, e.g., arithmetic functions like sin and some of the I/O procedures.

Despite the orthogonality and consistency of Algol 68, it contains several features not needed to satisfy TINMAN requirements (other than I5). Among these are: subroutine variables, alternate notations for case and conditional statement, formatted I/O, complex coercion rules, conditional and case expressions, array trimscripts (e.g., A(i:j@k)), valued assignment statements (e.g., X[(j := i)]), flexible arrays, the bits and bytes data types, etc. In short, Algol 68 contains more generality than the TINMAN requires (or permits).

I5. Simple Base Language (J3B)

Partially Satisfied.

Although J3B does not satisfy the concept of a kernel language with user-definable extensions, it is a simple language in terms of its variety of data types, control structures, and assumptions about its run-time environment. But to conform to the TINMAN requirements for simplicity, the absolute value function would have to be removed from the language (and be reintroduced via extension capabilities not presently available, since ABS is a generic function; i.e., it takes different types of arguments as its input parameter and returns a corresponding result type. Such properties of functions cannot be specified within J3B.), and the character string data type would have to be removed (so character strings could be supported as arrays of characters). It may also be that the J3B macro facility [Sections 2.2.6 and 3.4] should be dispensed with to satisfy I5. These, however, appear to be the only J3B capabilities that need to be deleted because of simplicity considerations. Other features, of course, need to be modified to conform to other requirements, and many features need to be added. But the fact that relatively few features violate I5's requirements for simplicity confirm the evaluation that J3B essentially satisfies I5.

I5. Simple Base Language (PASCAL)

Satisfied.

PASCAL, unlike ALGOL 68, is not usually thought of in kernel language/extended language terms. But its modest size (e.g., a self compiler consisted of only 4000 PASCAL source statements - Wirth, SPE, p. 321) and its data type definition facility permit it to be regarded as a reasonable base language. Certain procedures and functions are, of course, standard and are in effect predeclared in every implementation as are the two textfiles, input and output [pp. 160-167]. Particular definitions/extensions can be oriented to a specific environment and yet can be fully expressible within standard PASCAL. For example, PASCAL 6000 uses a predefined type "alfa" (which is a packed array of 10 characters) because on CDC 6000 series machines "a value of type alfa is representable in exactly one word" [p. 97].

I5. Simple Base Language (SIMULA 67)

Partially Satisfied.

The language is designed as a Common Base Language that defines the features common to all implementations. This base or "substrate" can be extended by defining new types with their accompanying operations [CBL 1.2]. That is, "SIMULA 67 may be oriented towards a special application area by defining a suitable class containing the necesary problem-oriented concepts. This class is then used as a prefix to the program by a user interested in the problems in question" [CBL 1.3.4]. However, the base language is not minimal. For example, the built-in type text provides for character strings and their manipulation [CBL 10]. This is done rather than synthesizing character strings from arrays of single characters. Indeed, character in SIMULA 67 is a distinct type from text [CBL 3.1].

I5. Simple Base Language (TACPOL)

Not Satisfied.

TACPOL is a reasonably small language and does have most numeric functions such as absolute value, maximum, minimum, and remainder as built-in procedures [Section A-2, A-3] instead of as language operators. But TACPOL has a character string data type [Section 3-3.a] with an infix catenation operator CAT [Section 6-1.c.(1)] rather than synthesizing these features out of arrays of single characters. Multiple target assignments are supported [Section 5-1.d]. This too violates I5's simple base requirement.

I6. Translator Restrictions (TINMAN)

Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

Limits on the number of array dimensions, the length of identifiers, the number of nested parentheses levels in expressions, or the number of identifiers in programs are determined by the translator and not by the object machine. Ideally, the limits should be set so high that no program (save the most abrasive) encounters the limits. They should be small enough that they do not dominate the compiler and large enough that they do not interfere with the usefulness of the language. If they were set at say the 99 percent level based on statistics from existing DoD computer programs the limits might be a few hundred for numbers of identifiers and less than ten in the other cases mentioned above.

I6. Translator Restrictions (ALGOL 68)

Not Satisfied.

Algol 68 does not define any translator limits. In Algol 68, identifier lengths, number of dimensions, nesting levels, number of identifiers, etc. may be arbitrarily large.

I6. Translator Restrictions (J3B)

Partially Satisfied.

Limits on the number of array dimensions are specified explicitly in the J3B language definition [Section 4.4.]. (Up to 3 dimensions are permitted.) Limits on identifier length, character string length, and fixed point scale factors are also specified [Sections 2.2.1 and 1.3.1]. Other limits on the size of program that can be compiled, the number of identifiers permitted, etc., are not specified, however.

I6. Translator Restrictions (PASCAL)

Not Satisfied.

Translator restrictions such as identifiers "must differ over their first 8 characters" and labels have "at most 4 digits" are stated as part of the definition for standard PASCAL [p. 168]. But some more important translator restrictions are not explicitly specified in the defining documents. For example, PASCAL 6000 produces error indicators for: too many cases in case statement, too many nested scopes, too many forward references, procedure code is too long, too many local files, and expression too complicated [pp. 120-121]. Standard PASCAL addresses none of these constraints.

I6. Translator Restrictions (SIMULA 67)

Not Satisfied.

The SIMULA 67 language definition does not specify such restrictions except for "not allowing an IF-statement to follow a THEN" [B 361]. For system classes the definition merely acknowledges that "an implementation may restrict the number of block levels at which such prefixes or block prefixes may occur in any one program" [CBL 14].

I6. Translator Restrictions (TACPOL)

Partially Satisfied.

Separating translator and object machine restrictions is not easy because the language reference manual defines the language as it is implemented on a single object machine. Only some translator restrictions, such as the number of array dimensions [Section 4-2] and identifier abbreviations [Section 2-6.g] besides maximum size for bit and character strings [Section 3-5.c,d], are made explicit. For example, maximum permissible nesting depth for blocks and expressions are not stated.

I7. Object Machine Restrictions (TINMAN)

Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

Limits on the amount of run time storage, access to specialized peripheral equipments, use of special hardware capabilities and access to real time clocks are dependent on the object machine and configuration. The translator will report when a program exceeds the resources or capabilities of the intended object machine but will not build in arbitrary limits of its own.

I7. Object Machine Restrictions (ALGOL 68)

Partially Satisfied.

Algol 68 does not define any object machine restrictions [Section 2.2]. However it does not prevent an implementation from doing so.

I7. Object Machine Restrictions (J3B)

Not Satisfied.

Language restrictions dependent on the object machine are explicitly specified as part of the J3B language definition [Section 1.3.1]. These include the maximum range of representable integer, floating, and fixed point values, the number of bits occupied by floating and fixed representations, the floating and fixed point precisions supported, the word length of the object computer (which affects the maximum bit string length permitted), the number of bits occupied by a pointer value, the number of bits occupied by a character, and the maximum number of words per record.

I7. Object Machine Restrictions (PASCAL)

Partially Satisfied.

No PASCAL restrictions are inherently dependent only on the object environment. But all program file processing (including all input and output) is restricted to sequential files [pp. 145, 161, 164-167]. So, a major restriction on the range of flexibility in using the object environment is defined into the language.

I7. Object Machine Restrictions (SIMULA 67)

Partially Satisfied.

SIMULA 67 does not provide for "access to real time clocks" nor for any parallel processing capability. These language restrictions limit the range

or the use of possible object environments. The class concept could be used to encapsulate access to real time clocks [CBL 1.3.4].

I7. Object Machine Restrictions (TACPOL)

Satisfied.

The TACPOL reference manual defines TACPOL as it is to be implemented for a particular computer, the AN/GYK-12. No concern is given for making TACPOL efficient on other computers. If efficiency considerations are ignored, TACPOL is quite free of object machine restrictions.

J1. Efficient Object Code (TINMAN)

The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

The language should not force programs to require greater generality than they need. When a program does not use a feature or capability it should pay no run time cost for the feature being in the language or library. When the full generality of a feature is not used, only the necessary (reduced) cost should be paid. Where possible, language features (such as, automatic and dynamic array allocation, process scheduling, file management, and I/O buffering) which require run time support packages should be provided as standard library definitions and not as part of the base language. The user will not have to pay time and space for support packages he does not use. Neither will there be automatic movement of programs or data between main storage and backing storage which is not under program control. Language features will result in special efficient object code when their full generality is not used. A large number of special cases should compile efficiently. If the base language components are easily composable they can be used to construct the specialized structures needed by specific applications, if they are simple and provide a single capability they will not force the use of unneeded capabilities in order to obtain needed capabilities, and if they are compatible with the features normally found in sequential uniprocessor digital computers with random access memory they will have near minimum or at least low cost implementation on many object machines.

J1. Efficient Object Code (ALGOL 68)

Not Satisfied.

Most of Algol 68 can be efficiently implemented[Section 0.1.4]. For example, the I/O and file management is all done by standard procedures [Section 10.3]. However, Algol 68 provides dynamic storage allocation (stack and garbage-collected) [Section 5.2.3] as part of the base language, along with nested recursive procedures [Section 5.4.1] and parallel processes [Section 3.3]. Garbage collection requires that additional information about

pointers and types be maintained at run time. In addition, string data (flexible arrays of characters) cannot be efficiently stored on a stack; even if a programmer wishes to specify an upper bound on string lengths, he cannot do so. Furthermore, because procedures can exit to a containing scope via a GOTO, with the possibility of terminating an unknown (at compile time) number of intervening subroutine calls, an efficient means of terminating a routine with an exit to its caller cannot be realized (i.e., implementing label parameters as procedures complicates an already complicated problem).

J1. Efficient Object Code (J3B)

Partially Satisfied.

J3B imposes a minimal burden in terms of run-time support packages, which consist primarily of subroutines for manipulating character and bit strings, exponentiation, and manipulation of table entries. J3B in its use for producing avionics applications programs has proven capable of producing sufficiently efficient object code for these applications. J3B compilers provide a number of register allocation and subroutine linkage conventions to permit efficient code to be generated.

There is no support provided for programmer control of program and data movement between main and backing store.

J1. Efficient Object Code (PASCAL)

Partially Satisfied.

The file handling and the dynamic (heap) allocation language features are provided as standard predeclared procedures [pp. 160-161]. Hence they can be implemented as run time support packages which are included only if needed. PASCAL does not support process scheduling or dynamic arrays. There is no program control of the automatic movement of programs or data between main store and backing store. PASCAL "was designed so that implementation of most of its constructs should be possible without a need for extensive optimization processes on conventional computers" [Wirth, SPE, p. 316].

J1. Efficient Object Code (SIMULA 67)

Not Satisfied.

SIMULA 67 does not provide for programmer control over overlays. Garbage collection occurs automatically and cannot be invoked by the programmer [CBL 9.1]. The SIMULA 67 language and compilers have been designed so that "only those runtime routines needed are included" [Structure, p. 6]. Class concatenation yields, in effect, variant record structures that occupy more space than if the programmer could specify the record structure directly, because type discrimination fields are included in the record and their position and coding are not under program control (Palme, J., "List structures

in SIMULA and PL/I -- a comparison." Software- Practice and Experience, volume 4, no. 4 (Oct.-Dec. 1974), pp. 379-399). Moreover, since records can only be heap allocated, time and space must be taken at run time to create class instances that could, as far as the programmer is concerned, be allocated and initalized prior to run time. Text data must be allocated on the heap (even for what would be local variables in another language) because of the ability to assign strings of arbitrary length to the same text variable.

J1. Efficient Object Code (TACPOL)

Partially Satisfied.

"TACPOL has been designed to restrict the use of those source language elements which could easily generate large volumes of machine language code" [Section 1-6]. As a result, the programmer "does not have to be concerned with implicitly generating large volumes of machine language code." TACPOL does make it possible to generate efficient object code. It provides partial program control over the movement of programs or data between main and backing store with the LOAD statement [Section 13-15] whose execution causes a designated program to be loaded into memory.

J2. Safe and Effective Optimization Possible (TINMAN)

Any optimizations performed by the translator will not change the effect of the program.

The number of applicable safe optimizations can be increased by making more information available to the compiler and by choosing language constructs which allow safe optimizations. This requirement allows optimization by code motion providing that motion does not change the effect of the program.

J2. Safe and Effective Optimization Possible (ALGOL 68)

Partially Satisfied.

The Algol 68 Revised Report defines explicitly and precisely what action, effects, and properties must be satisfied by an implementation [Section 2.2]. All else is left open to possible optimization. But collateral elaboration of expressions (see C1) permits optimizations to change the result of executing a program, and lack of adequate information about a subroutine's possible side-effects prohibits some optimizations that would be safe if a compiler had additional information. On the other hand, the requirement that pointers be typed permits some optimization that would otherwise be impossible.

J2. Safe and Effective Optimization Possible (J3B)

Partially Satisfied.

Because the order in which operands of arithmetic expresssions are evaluated is not fully specified [Section 7.1], optimizations can change the effect of a program and still satisfy the language specification. Similarly, since short circuit evaluation of Boolean expresssions is explicitly permitted, but not required [Section 7.3], the results of performing or not performing short circuit evaluation can change the effect of program executions in some cases and still satisfy the language specification.

The provision of typed pointers, on the other hand, permits more effective code optimization because an assignment through a pointer can modify only a restricted set of locations. Consequently, the compiler can determine that the contents of certain registers do not need to be updated after the asssignment. Similarly, the unsigned integer data type provides implicit range information to a compiler that can be used in optimizing programs.

J2. Safe and Effective Optimization Possible (PASCAL)

Satisfied.

Wirth has co-authored with C.A.R. Hoare an axiomatic definition of PASCAL which requires that language features have their defined meaning regardless of implementation and optimization approaches. The packed prefix [p. 43], the subrange type [p. 143], and the dispose procedures [p. 161] are examples of language features which present opportunities for, but do not require, (storage) optimizations. But this does not ensure that an optimized program will produce unchanged effects if the difference in effect is permitted by the language definition. In this sense, PASCAL does not satisfy J2 completely, since (for example) operands in expressions may be evaluated in different orders, even if some operands are functions having side effects, and this can lead to different (but valid) results. On the other hand, code motion eliminating some function calls cannot generally be safely performed because a compiler does not have adequate information about the subroutine's possible side effects.

J2. Safe and Effective Optimization Possible (SIMULA 67)

Partially Satisfied.

"The Common Base comprises the language features required in every SIMULA 67 compiler" [CBL Preface]. Optimizations are not directly addressed in the defining document, but by inference any optimizations are required to conform to the Common Base Language semantics. But since SIMULA does not constrain the order in which operands of expressions are evaluated (see C1), optimizations yielding different evaluation orderings can produce different results and still conform to the language specifications. Similarly, lack of adequate information about a subroutine's possible side effects prohibits some optimizations that would otherwise be safe.

J2. Safe and Effective Optimization Possible (TACPOL)

Not Satisfied.

For example, the default packing of structures - groups, tables, cells - [Section 4-4] can be changed by using the ALIGNED option that optimizes access at the expense of storage. This packed or aligned choice does not affect the rest of the program code except if the structure is copied with the MOVE operator, which does not check packing (or other structural attributes) for consistency between the source and target of the MOVE operator.

J3. Machine Language Insertions (TINMAN)

The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

The ability to enter machine language should not be used as an excuse to exclude otherwise needed facilities from the HOL; the abstract description of programs in the HOL should not require the use of machine language insertions. The semantics of machine language insertions will be determinable from the HOL definition and the object machine description alone and not be dependent on the translator characteristics Machine language insertions will be encapsulated so they can be easily recognized and so that it is clear which variables and program identifiers are accessed within the insertion. The machine language insertions will be permitted only within the body of compile time conditional statements (see I4) which depend on the object machine configuration (see I3). They will not be allowed interspersed with executable statements of the source language.

J3. Machine Language Insertions (ALGOL 68)

Not Satisfied.

Algol 68 does not define a mechanism for machine language insertions, nor for compile time conditional statements. An implementation may use pragmats [Section 9.2] for this purpose, however.

J3. Machine Language Insertions (J3B)

Not Satisfied.

J3B makes no provision for incorporating assembly code into programs, either directly or in encapsulated form, although an implementation may support a special set of functions that give access to some machine dependent capabilities, including specific machine instructions [Section 3.2.1]. For example, HOL programmers can be given access to machine level I/O instructions in this way. But this method of providing access to assembly code is not as general or as disciplined as the TINMAN requires.

J3. Machine Language Insertions (PASCAL)

Not Satisfied.

Machine language code insertions and compile time conditional statements are not part of PASCAL and they violate the language's design philosophy [p. 136].

J3. Machine Language Insertions (SIMULA 67)

Not Satisfied.

Not supported in the SIMULA 67 Common Base Language.

J3. Machine Language Insertions (TACPOL)

Partially Satisfied.

TACPOL allows machine language insertions via CODE blocks. The machine code is bracketted by CODE and END particles. "Names defined in TACPOL blocks are not normally known to CODE blocks" [Section 7-5.a]. The USES particle followed by a list of previously defined names "marks the names in the list known in the CODE block" [Section 7-5.b]. However, these CODE blocks do not appear within the body of compile time conditional statements. Moreover, examination of some actual TACPOL program shows that the USES particle is not actually required in at least some cases.

J4. Object Representation Specifications (TINMAN)

It will be possible within the source language to specify the object representation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

It will be possible to specify the order of the fields, the width of fields, the presence of don't care fields, and the position of word boundaries. It will be possible to associate source language identifiers (data or program) with special machine addresses. The use of machine dependent characteristics of the object representation will be restricted as with machine dependent code (see J3).

If the object representation of a composite data object is not specified in the source program, there will be no specific default guaranteed by the translator.

J4. Object Representation Specifications (ALGOL 68)

Not Satisfied.

Algol 68 does not define a mechanism for specifying object representations of data. The programmer can only specify logical representations [Section 4.6]. There are no packed or aligned attributes. However, an implementation may use pragmats [Section 9.2] for this purpose. Algol 68 leaves the choice of object representation to the translator.

J4. Object Representation Specifications (J3B)

Partially Satisfied.

J3B's "specified" table construct permits programmers to specify the object representation of records and arrays of records (i.e., tables). However, the syntax for specifying this representation is not separated from the description of the structure's logical properties [Sections 4.3 and 4.5].

Similarly, the packing specifications supported for "ordinary" table declarations [Section 4.3] permit a programmer to specify space/time tradeoffs in a general way, leaving the specific organization of a record's fields to the compiler.

Source language identifiers cannot be associated machine addresses.

The BIT and BYTE operators provide access to the object machine representation of any data value. Dependence on object machine representation is indicated when these operators are applied to other than bit or character strings, respectively.

It is not possible to specify the presence of "don't care" fields.

The serial and parallel forms of table structure are generic ways of specifying the object representation of table [Section 4.2].

J4. Object Representation Specification (PASCAL)

Not Satisfied.

The PASCAL programmer cannot specify the object representation of structured types. The prefix "packed" signifies a wish to economize on space at the expense of time [p. 143]. But this wish might be ignored by the implementator.

Object machine addresses cannot be associated with program identifiers.

J4. Object Representation Specification (SIMULA 67)

Not Satisfied.

There is no provision in SIMULA 67 for specifying object representations but the class concept, if extended to include this capability, would serve to encapsulate or "hide" the machine dependencies.

J4. Object Representation Specifications (TACPOL)

Partially Satisfied.

The TACPOL user has the (relatively) machine independent choice of PACKED or ALIGNED for the object representation of composite data structures [Section 4-4]. The choice applies to the entire structure and cannot be used more selectively. Thus, specification at the level of word boundaries for components of PACKED structures is not possible. Nor can special machine addresses be associated with source language identifiers.

J5. Open and Closed Routine Calls (TINMAN)

The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.

Open routine capability is especially important for machine language insertions.

The distinction between open and closed implementation of a routine is an efficiency consideration and should not affect the function of the routine. Thus, an open routine will differ from a syntax macro in that (a) its global environment is that of its definition and not that of its call and (b) multiple occurrences of a formal value (i.e., read only) parameter in the body have the same value. If a routine is not specified as either open or closed the choice will be optimal as determined by the translator.

J5. Open and Closed Routine Calls (ALGOL 68)

Not Satisfied.

Algol 68 does not define a way to specify whether calls of a routine are to be open or closed. However, an implementation may use pragmats [Section 9.2] for this purpose.

J5. Open and Closed Subroutine Calls (J3B)

Satisfied.

The J3B INLINE directive, when applied to the name of a subroutine, specifies to a compiler that that subroutine is to be compiled in open form [Section 3.3.4]. If actual parameters are constants, and the substitution of actual for formal parameters yields expressions that are evaluable at compile time in J3B (see C4) [Section 8], these expressions are evaluated before object code is generated. Except for efficiency effects, the INLINE and closed form of a subroutine are required to have the same semantic effect [Section 3.3.4].

J5. Open and Closed Routine Calls (PASCAL)

Not Satisfied.

The PASCAL programmer cannot specify whether calls will have an open or closed implementation. This important capability is left exclusively to the translator. Adding this feature to the language would not impact the other features.

J5. Open and Closed Routine Calls (SIMULA 67)

Not Satisfied.

The programmer cannot specify an open or closed implementation for routine calls.

J5. Open and Closed Routine Calls (TACPOL)

Not Satisfied.

The TACPOL programmer cannot specify whether procedure calls have an open or closed implementation.

K1. Operating System Not Required (TINMAN)

The language will not require that the object machine have an operating system. When the object machine does have an operating system or executive program, the hardware/operating system combination will be interpreted as defining an abstract machine which acts as the object machine for the translator.

K1. Operating System Not Required (ALGOL 68)

Partially Satisfied.

Algol 68 does not require an operating system. However, some of its features, e.g. I/O, need run time support routines drawn from a library.

K1. No Operating System Required (J3B)

Satisfied.

J3B programs make no demands on operating system capabilities except that the file names referenced in COMPOOL and COPY directives must conform to the length and identifier conventions of the host operating sytem [Sections 3.2.2.5 and 3.1.1]. This requirement, however, does not impact the semantics of object programs, and so does not violate K1.

K1. Operating System Not Required (PASCAL)

Satisfied.

Program input and output using file data types requires buffering and administration of secondary storage [pp. 145, 161, 164-167], but this does not necessitate a full operating system. Interestingly, in the initial compiler implementation effort, interfacing with the operating system was one of the three principal development tasks [Wirth, SPE, p. 323].

K1. Operating System Not Required (SIMULA 67)

Partially Satisfied.

None of SIMULA 67's features per se require an operating system. However, the language definition does assume the compiler exists in an operating system environment. For example, under the separate compilation category it is noted that "an external identifier is an identification with respect to an 'operating system' of a separately compiled declaration" [CBL 15.2].

K1. Operating System Not Required (TACPOL)

Satisfied.

TACPOL has no feature which would require a full operating system to support; no mention of a supporting operating system is made in the language reference manual. TACPOL is intended, in part, to help develop and maintain the operating system for TACFIRE software [Section 1-2.b]. On the other hand, the discussion of I/O mentions partitioned data sets, a capability not supported in all environments TACPOL might be used in. The LABELLED particle [Section 14-3] specifies a "file is to be processed with standard header and trailer labels." This appears to presume the existence of operating system conventions, at least. Moreover, the OPEN statement [Section 13-4] specifies whether a file is NEW, OLD, KEEP, or PASS, which appears to presume the particular file management capabilities of a well known operating system.

K2. Program Assembly (TINMAN)

The language will support the integration of separately written modules into an operational program.

Separately written modules in the form of routines and type definitions are necessary for the management of large software efforts and for effective use of libraries. The user will be able to cause anything in any accessible library to be inserted into his program. This is a requirement for separate definition but not necessarily for separate compilation. The decision as to whether separately defined program modules are to be maintained in source or object language form is a question of implementation efficiency, will be a local management option and will not be imposed by the language definition. The trade-offs involved are complicated by other requirements for type checking of parameters (see C6), for open subroutines (see J5), for efficient object representations (see J1), and for constant expression evaluation at compile time (see C4).

K2. Program Assembly (ALGOL 68)

Not Satisfied.

Algol 68 does not explicitly define a mechanism for integrating separately written modules, although an implementation may use pragmats [Section 9.2] for copying code from a library or for specifying separate compilation, etc. For example,

pr copy filename from libraryname pr

K2. Program Assembly (J3B)

Satisfied.

J3B supports (but does not require) the separate compilation of programs [Section 3.1], and the COPY directive can be used to insert source language modules into programs being compiled [Section 2.2.5]. The COMPOOL directive is used to access files of declarations for use in compiling a program [Sections 3.1.2 and 3.1.3]. The two directives together suffice to cause anything in an accessible "library" to be inserted into a program. In addition, COMPOOL declarations of subroutines are checked against the actual subroutine definition for accuracy [Section 3.3.2, p. 3-18, errors 8 and 9].

K2. Program Assembly (PASCAL)

Not Satisfied.

No language features support the combining of independently written modules. The original PASCAL compiler, which has served as the basis for subsequent ones, used a one-pass scheme and generated absolute code.

Compilation speed was gained by avoiding a relocating loader. Wirth believes that "merging of programs should occur at the source language level" [Wirth, SPE p. 317]. However, PASCAL 6000 allows the integration of object code modules as called procedures or functions [p. 101]. A constraint on separately written type definitions is that a given identifier can appear in at most one enumeration of a scalar type [p. 34].

K2. Program Assembly (SIMULA 67)

Partially Satisfied.

The SIMULA 67 definition makes separate compilation optional rather than mandatory. "If an implementation permits user defined procedure and class declarations to be separately compiled, then a program should have means of making reference to such declarations as external to the program" [CBL 15]. But support for integration of modules is not provided as a mandatory part of the language.

K2. Program Assembly (TACPOL)

Satisfied.

Through the integration of separately defined program modules is not addressed in the TACPOL manual, the language has a Compool capability. The Compool contains programs and data "that are commonly used by many different programs in a system" [Section 14-1].

K3. Software Development Tools (TINMAN)

A family of programming tools and aids in the form of support packages including linkers, loaders and debugging systems will be made available with the language and its translators. There will be a consistent easily used user interface for these tools.

K3. Software Development Tools (ALGOL 68)

Not Determinable.

The Algol 68 Revised Report does not discuss software tools. The language is appropriate for constructing such tools. Some of the current Algol 68 compilers are coded in Algol 68.

K3. Software Development Tools (J3B)

Not Determinable.

No software development tools are specified within the J3B language specification.

K3. Software Development Tools (PASCAL)

Not Determinable.

Defining documents do not address the tools requirement. Compilers generating absolute code (as most present PASCAL compilers do) are not strictly compatible with linkers and loaders [Wirth, SPE, p. 317]. Wirth downplays debugging systems, favoring instead a programming language that facilitates program formation with automatic (largely compile-time) consistency checks [Wirth, p. 197].

K3. Software Development Tools (SIMULA 67)

Not Determinable.

Although a representative SIMULA 67 implementation includes diagnostic routines and additional routines for user convenience [Structure, p. 6], such tools are not part of the language specification. In particular, "a consistent easily used user interface" is not defined.

K3. Software Development Tools (TACPOL)

Not Determinable.

The TACPOL documentation does not specify support tools to be made available with the language. In fact, TACPOL was designed "for use in developing the TACFIRE software" [Section 1-1]. This includes a range of aids and diagnostic programs [Section 1-2].

K4. Translator Options (TINMAN)

> A variety of useful options to aid generation, test, documentation and modification of programs will be provided as support software available with the language or as translator options. As a minimum these will include program editing, post-mortem analysis and diagnostics, program reformatting for standard indentations, and cross-reference generation.

There will be special facilities to aid the generation, test, documentation and modification of programs. Tools and debugging aids will be source language oriented.

Some of the translator options which have been suggested and may be useful include the following. Code might be compiled for assertions which would give run time warnings when the value of the assertion predicate is

false. It might provide run time tracing of specified program variables. Dimensional analysis might be done on units of measure specifications. Special optimizations might be invoked. There might be capability for timing analysis and gathering run time statistics. There might be translator supplied feedback to provide management visibility regarding progress and conformity with local conventions. The user might be able to inhibit code generation. There might be facilities for compiling program patches and for controlling access to language features. The translator might provide a listing of the number of instructions generated against corresponding source inputs and/or an estimate of their execution times. It might provide a variety of listing options.

K4. Translator Options (ALGOL 68)

Partially Satisfied.

Algol 68 does not define any translator options. It does allow an implementation to use pragmats [Section 9.2] to specify options, e.g.

pr crossref, attributes, trace, object list pr

K4. Translator Options (J3B)

Not Determinable.

Translator options are not specified in the J3B language specification, although actual J3B compilers provide cross reference listings and a variety of optimization options, including register allocation strategies and linkage conventions.

K4. Translator Options (PASCAL)

Not Determinable.

Defining documents do not specify standard translator options, but PASCAL 6000, for example, offers options to include important run-time tests, to obtain a complete post-mortem dump that is source code oriented, etc [pp. 100-103]. Users have complained about "the spartan compilation listing produced by the compiler" [PASCAL Newsletter, No. 3, February 1975, SIGPLAN Notices, 11, 2, February 1976, p. 38].

K4. Translator Options (SIMULA 67)

Not Determinable.

No translator options are suggested or required for SIMULA 67's Common Base Language. However, standard SIMULA 67 compilers often produce cross reference listings [Structure, p. 4] and provide "several options to improve the printout of the source program" [Structure, p. 5].

K4. Translator Options (TACPOL)

Partially Satisfied.

TACPOL provides condition declarations as a debugging aid [Section 12-1]. Conditions that arise and have been declared by CHECK produce a diagnostic snapshot. Besides zerodivide and fixed overflow checking, extensive execution tracing can be done for assignments, procedure calls, and GOTOs by specifying the appropriate names with the USAGE particle [Section 12-2]. A TACPOL compiler outputs an attribute and reference list, cross reference listings, and set/used information besides source and object code [Section D-1]. Editors, formatters, and post- mortem programs are not mentioned.

K5. Assertions and Other Optional Specifications (TINMAN)

The source language will permit inclusion of assertions, assumptions, axiomatic definitions of data types, debugging specifications, and units of measures in programs. Because many assertional methods are not yet powerful enough for practical use, nor sufficiently well developed for standardization, they will have the status of comments.

The language will not prohibit inclusion of these forms of specification if and when they become available for practical use in programs. Assertions, assumptions, axiomatic definitions and units of measure in source language programs should be enclosed in special brackets and should be treated as interpreted comments -- comments which are delimited by special comment brackets and which may be interpreted during translation or debugging to provide units analysis, verification of assertions and assumptions, etc. -- but whose interpretation would be optional to translator implementations.

K5. Assertions and Other Optional Specifications (ALGOL 68)

Satisfied.

Algol 68 provides a pragmat mechanism [Section 9.2] which meets this requirement exactly. Pragmats are comments, but are delimited by pr instead of co so that a translator may easily recognize them and optionally treat them as compile time directives, in particular to specify assertions.

K5. Assertions and Other Optional Specifications (J3B)

Not Satisfied.

There is no provision for special-comment brackets for including assertions or units measures, etc. in programs in a manner that can be ignored by translators not prepared to process these constructs.

K5. Assertions and Other Optional Specifications (PASCAL)

Not Satisfied.

There is no provision for special comment brackets that might be needed to delimit assertions. Extending PASCAL to meet requirement K5 would be straightforward, however. Moreover, all of the literature on the language (e.g., Wirth's Systematic Programming book) extensively uses assertions as comments. There is no provision for incorporating or making use of units of measure.

K5. Assertions and Other Specifications (SIMULA 67)

Not Satisfied.

No provision is made for optionally interpretable comments delimited by special brackets.

K5. Assertions and Other Optional Specifications (TACPOL)

Not Satisfied.

TACPOL does not support assertions or any form of interpretable comments.

L1. No Superset Implementations (TINMAN)

No implementation of the language will contain source language features which are not defined in the language standard. Any interpretation of a language feature not explicitly permitted by the language definition will be forbidden.

This does not, however, disallow library definition optimizations which are translator unique.

L1. No Superset Implementations (ALGOL 68)

Not Satisfied.

The Algol 68 Revised Report precisely defines the language and what requirements must be satisfied to have an "implementation" be an implementation of Algol 68. However, it permits features to be added by an implementation as long as all standard Algol 68 programs retain the same syntax and semantics [Section 2.2].

L1. No Superset Implementations (J3B)

Not Satisfied.

J3B permits implementation-dependent "built-in" functions to be defined [Section 3.2.1]. These language constructs have the syntactic appearance of functions, but they may have special properties not definable within the language, or they may provide access to special object machine instructions, such as I/O instructions. Programs written using these functions cannot be compiled by a compiler that does not support "built-in" functions having identical names and properties.

L1. No Superset Implementations (PASCAL)

Satisfied.

The definition of PASCAL establishes a standard that excludes further source language features.

L1. No Superset Implementations (SIMULA 67)

Not Satisfied.

Separate compilation is "an optional part of the Common Base" [CBL 15]. Consequently SIMULA 67 implementations with this facility are permissible supersets.

L1. No Superset Implementations (TACPOL)

Not Satisfied.

The TACPOL manual does not forbid superset implementations.

L2. No Subset Implementations (TINMAN)

Every translator for the language will implement the entire base language. There will be no subset implementations of the base language.

The intended source language product from this effort is a small, simple, uniform, base language with specialized features, support packages, and complex features relegated to library routines not requiring direct translator support. If simple low cost translators are not feasible for the selected language, then the language is too large and complex to be standardized and the goal of language commonality will not be achievable.

L2. No Subset Implementations (ALGOL 68)

Not Satisfied.

The Algol 68 Revised Report precisely defines the language and what requirements must be satisfied to have an "implementation" be an implementation of Algol 68. However, it permits features to be deleted by an implementation as long as all subset programs have the same syntax and semantics as in a full Algol 68 implementation [Section 2.2]. In fact, most existing Algol 68 implementations omit the parallel processing feature and some omit garbage collection of the heap.

L2. No Subset Implementations (J3B)

Not Satisfied.

The language specification permits some implementations to not support the fixed point and double precision floating point data types [Sections 7.1.3 and 7.1.4]. If floating point data are not supported, then the exponentiation operator is also not supported [Sectin 7.1]. In addition, all implementations are not required to support unaligned pointers [Section 1.3.3.1, p. 1-16].

L2. No Subset Implementations (PASCAL)

Satisfied.

The defining documents establish standard PASCAL without allowing for subsets. (The first PASCAL compiler for the ICL 1900 series of computers (SPE, vol. 2, no. 2, pp. 75-76), however, did not implement set operations.)

L2. No Subset Implementations (SIMULA 67)

Satisfied.

The defined Common Base Language "is required to be a part of any SIMULA 67 system" [CBL 1.4].

L2. No Subset Implementations (TACPOL)

Not Satisfied.

The reference manual does not explicitly require a common baseline for all TACPOL implementations. In fact, at a syntactic level, restricted character sets (e.g, without a colon or a semicolon [Section 2-4]) are accommodated.

L3. Low Cost Translation (TINMAN)

The translator will minimize compile time costs. A goal of any translator for the language will be low cost translation.

Where practical and beneficial the user will have control over the level of optimization applied to his programs. The programmer will have control over the tradeoffs between compile time and run time costs. The goal will be effective use of the available machines both in object execution and translation and not maximal speed of translation.

Both the translator and the language design will emphasize low cost translation, but in an environment of large and long-lived software products this will be secondary to requirements for reliability and maintainability. Language features will be chosen to ensure that they do not impose costs for unneeded generality and that needed capabilities can be translated into efficient object representations.

L3. Low Cost Translation (ALGOL 68)

Partially Satisfied.

Algol 68 is more difficult to translate than a small language like PASCAL. Although Algol 68 is comparable in size and features to PL/I, Algol 68's orthogonality and choice of features permits a smaller, simpler compiler than PL/I does. In particular, Algol 68 is easier to optimize than PL/I [Section 0.1.4].

L3. Low Cost Translation (J3B)

Satisfied.

Certain J3B constructs were designed to reduce the cost of translation, in particular, the requirement that declarations be grouped at the beginning of lexical scopes [Section 3.1.1], the requirement that identifiers, in general, be declared before they are used [Sections 3.2.2, 3.2.3, and 3.2.2.4], and the provision for preprocessed compools and lexical (rather than character) macros [Sections 3.1.3 and 3.4].

L3. Low Cost Translation (PASCAL)

Satisfied.

Many language features were deliberately omitted from PASCAL in order to attain compilation speed that eliminated the need for relocatable object code [Wirth, SPE, p. 317]. "A rigorous selection among the immense variety of programming facilities available had to be made in order to keep the compiler relatively compact and efficient and economical for both the user who writes only small programs using few constructs of the language and the user who

writes large programs and tends to make use of the full language" [p. 8]. The syntax supports a one-pass scheme which greatly facilitates compilation speed on computers with large main storage.

L3. Low Cost Translantion (SIMULA 67)

Partially Satisifed.

Bench-mark tests have shown that the SIMULA 67 "IBM system compiles roughly twice as fast as PL/I-F" and on the Univac 1100 series SIMULA 67 compiles "about the same as NU-ALGOL". "I/O is significantly faster in SIMULA than in FORTRAN" [Structure, p. 11]. But no mention is made of user control over the level of optimization applied to programs.

According to J. Palme: "Most of the compilers are very good and efficient. For example, the IBM 360/370 compiler is about as efficient as the FORTRAN G compiler, and the CDC 3300 compiler is regarded as the best compiler existing for that computer" [Palme, p. 25].

L3. Low Cost Translation (TACPOL)

Satisfied.

The TACPOL language features have been chosen so that they have natural counterparts in patterns of assembly code. The costly and difficult-to-translate features of PL/I have been excised.

L4. Many Object Machines (TINMAN)

Translators will be able to produce code for a variety of object machines. The machine independent parts of translators might be built independent of the code generators.

L4. Many Object Machines (ALGOL 68)

Satisfied.

The Algol 68 Revised Report does not prohibit a translator from producing code for several object machines. Pragmats [Section 9.2] may be used to specify the desired object machine. Several Algol 68 translators already exist and/or are under construction.

L4. Many Object Machines (J3B)

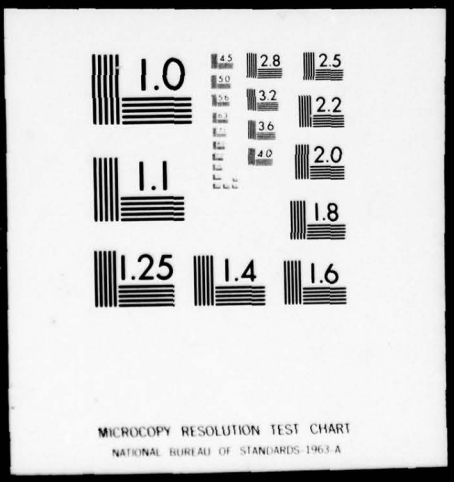
Satisfied.

AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

11 OF 28
ADA037634

11 OF 28
ADA037634



J3B compilers have been developed that generate code for the IBM 360/37 Delco M362F, Litton 516, and Singer SKC-2000.

L4. Many Object Machines (PASCAL)

Satisfied.

PASCAL compilers currently exist for the following computers: CDC 6000 series, IBM 360/370 series, ICL 1900 series, DEC 10, CII IRIS 80, Univac 110 series, CDC 3300, CDC 3600, B6700, PDP-11/20. A recent survey indicated tha PASCAL implementation "projects were underway for the following machines: TI ASC, Data General 840, Raytheon 704, Univac AN/UYK-20, Honeywell G635, Burroughs 4700 and 6700, and PDP-11/45" [PASCAL Newsletter in SIGPLAN Notice 11, 2, February 1976, p. 37].

L4: Many Object Machines (SIMULA 67)

Satisfied.

Already a number of implementations exist. J. Palme writes: "SIMULA systems are available on a large number of computers: IBM 360/370, Univac 1106-1110, CDC 3200-3500, CDC 3600, CDC 6400-6600, CDC Cyber 70 series and t larger CII and Iris computers. New SIMULA systems are under development for the ... DEC system 10 series" [Palme, p. 25].

L4: Many Object Machines (TACPOL)

Not Satisfied.

In principle, TACPOL could satisfy the L4 requirement. In fact, TACPOL (according to the reference manual) has been built for a particular machine, the AN/GYK-12.

L5. Self-Hosting Possible (TINMAN)

The translator need not be able to run on all the object machines. Self-hosting is not required, but is often desirable.

The size of machines which can host translators should be kept as small as possible by avoiding unnecessary generality in the language.

L5. Self-Hosting Not Required (ALGOL 68)

Not Determinable.

Whether Algol 68 can be hosted on a small computer is not determinable from information available to us.

L5: Self-Hosting Possible (J3B)

Satisfied (?)

Current J3B compilers are large because no design constraints were imposed to insure a small compiler would be produced. There appears to be no reason, however, why a small J3B compiler could not be developed.

L5: Self-Hosting Possible (PASCAL)

Satisfied.

The modest size of a self compiler (i.e., 4000 PASCAL source statements - Wirth, SPE, p. 321) permits a wide range of hosting machines.

L5: Self-Hosting Possible (SIMULA 67)

Partially Satisfied.

SIMULA 67 compilers are not excessively large; nor are they small. "The Univac compiler takes about 30 K words on the 1108" [Structure, p. 3] and the next release of the 360/370 version requires about 93K bytes.

L5: Self-Hosting Possible

Satisfied.

This assumes the L5 requirement constrains the compiler's potential size to fit on small, or at least medium, sized machines. TACPOL meets the L5 requirement in the above sense.

L6: Translator Checking Required (TINMAN)

The translator will do full syntax checking, will check all operations and parameters for type compatibility and will verify that all language imposed semantic restrictions on the source programs are met. It will not automatically correct errors detected at compile time.

L6: Translator Checking Required (ALGOL 68)

Satisfied.

The Algol 68 Revised Report precisely and completely specifies the syntax and semantics of the language. In particular, an Algol 68 program must have compatible operator-operand types and actual-formal parameter types. The Report requires an implementation to meet these requirements [Section 2.2].

L6. Translator Checking Required (J3B)

Satisfied.

Full syntax checking, including checks on parameter declarations and usage, is supported by every J3B compiler. No errors are automatically corrected v

L6. Translator Checking Required (PASCAL)

Not Determinable.

PASCAL was designed to facilitate full automatic checking for syntactic consistency [Wirth, p. 197], but the defining documents do not require it nor do they specify the kinds of errors to be detected.

L6. Translator Checking Required (SIMULA 67)

Partially Satisfied.

SIMULA 67 is designed for extensive compile time checking, The defining document [CBL] does not specify how much is required. In SIMULA compilers, all language errors "are discovered as soon as they logically can be discovered, and the error message is always phrased in SIMULA terms" [Palme, p. 30]. "The structure of the SIMULA language is such that almost all the error checking of a program can be done at compile time" [Palme, p. 38].

L6. Translator Checking Required (TACPOL)

Satisfied.

In general, "attributes must be explicitly declared. This results in fewer errors because missing specifications are flagged at compile time" [Section 1-6]. The TACPOL compiler is charged with full syntax checking. For example, it must detect and reject expressions with mixed mode.

L7. Diagnostic Messages (TINMAN)

The translator will produce compile time explanatory diagnostic error and warning messages. A suggested set of error and warning situations will be provided as part of the language definition.

Diagnostic messages will not be coded but will be explanatory and in source language terms. Translators will continue processing and checking after errors have been found but should be careful not to generate erroneous messages because of translator confusion. The translator will always produce correct code; when source program errors are encountered by the translator or referenced program structures omitted, the compiler will produce code to cause

a run time exception condition upon any attempt to execute those parts of the program. Warnings will be generated when a source language construct is exceptionally expensive to implement on the specified object machine. A suggested set of diagnostic messages, provided as part of the language definition, contributes to commonality in the implementation and use of the language.

L7. Diagnostic Messages (ALGOL 68)

Not Satisfied.

The Algol 68 Revised Report does not define translator actions for erroneous programs, nor does it provide a suggested set of error and warning situations.

L7. Diagnostic Messages (J3B)

Partially Satisfied.

Some J3B program errors to be detected by compilers are specified implicitly in the J3B syntax equations, i.e., a program not conforming to the equations is considered to be in error. Error situations not specified in the syntax equations are specifically listed in the language specification. These situations are divided into two classes, "Illegal Constructs" (errors that are detectable at compile time), and "Undefined Effects" (errors detectable only at run time). Specific constructs that are legal, but likely to be indicative of programmer error, are also explicitly cited in the specification as "Warnings". The J3B specification states, "It is expected that all violations of the syntax equations, all 'Illegal Constructs' and all 'Warnings' will be detected by every compiler implementation for JOVIAL/J3B, although strictly speaking, which errors will be detected is defined by the compiler implementation specification" [Section 1.3.1]. Whether or not code is generated to check for "Undefined Effects" is implementation-dependent, and is generally a possible compiler option. The language specification does not demand that such code be generated.

Although error situations are completely documented, the language specification does not specify or suggest a set of error messages, nor does the specification limit a compiler's behavior once an error has been found. In practice, the J3B compilers use a common set of error messages and produce no code if compile-time errors are detected.

L7. Diagnostic Messages (PASCAL)

Not Satisfied.

Such messages are left to the PASCAL implementor. The PASCAL 6000 error messages [pp. 119-121] are a reasonable set of diagnostics that could be suggested as part of the standard language.

L7. Diagnostic Messages (SIMULA 67)

Not Satisfied.

No suggested set of diagnostic messages are provided as part of the language definition.

L7. Diagnostic Messages (TACPOL)

Not Satisfied.

Whereas in practice TACPOL does give diagnostic error and warning messsges, the language reference manual does not list nor suggest any such messages.

L8. Translator Internal Structure (TINMAN)

The characteristics of translator implementations will not be dictated by the language definition or standards.

L8. Translator Internal Structure (ALGOL 68)

Satisfied.

The Algol 68 Revised Report does not define the internal characteristics of translators and/or run time support routines.

L8. Translator Internal Structure (J3B)

Satisfied.

The J3B definition avoids dictating compiler charactistics.

L8. Translator Internal Structure (PASCAL)

Satisfied.

The PASCAL definition avoids dictating compiler characteristics.

L8. Translator Internal Structure (SIMULA 67)

Satisfied.

The language definition does not constrain unduly a translator's internal structure.

L8: Translator Internal Structure (TACPOL)

Satisfied.

The TACPOL manual does not recommend or restrict implementation techniques.

L9: Self-Implementable Language (TINMAN)

Translators for the language will be written in their own source language.

L9: Self-Implementable Language (ALGOL 68)

Satisfied.

Algol 68 is a suitable language for writing an Algol 68 translator. At least one such translator exists.

L9: Self-Implementable Language (J3B)

Partially Satisfied.

J3B's capabilities are sufficient and reasonable for writing a compiler in J3B, but to date, this has not been done for any J3B compiler. (All J3B computers are written in AED and run on IBM 360/370 computers.)

L9: Self-Implementable Language (PASCAL)

Satisfied.

The first PASCAL compiler was written in FORTRAN and then discarded. The second compiler was originally written in PASCAL and then translated by hand. PASCAL has carried self-compilation to the extreme that currently there is no compiler "available in any other language than PASCAL itself" [Wirth, SPE, p. 324].

L9: Self-Implementable Languge (SIMULA 67)

Partially Satisfied.

Though SIMULA 67 has not strictly been self-implemented, a standard "abstract implementation was given in a slightly extended SIMULA language, and covered most of the support routines required during execution of user programs... This abstract definition caused most SIMULA runtime systems to use similar techniques, and simplified both the implementation and the debugging of the different runtime systems" [Structure, p. 2].

L9. Self-Implementable Language (TACPOL)

Partially Satisfied.

TACPOL has not yet been implemented in itself. But TACPOL was designed, in part, to help in developing and in maintaining compilers [Section 1-2.c].

M1: Existing Language Features Only (TINMAN)

> The language will be composed from features which are within the state of the art and any design or redesign which is necessary to achieve the needed characteristics will be conducted as an engineering design effort and not as a research project.

The adoption of a common language can be successful only if it makes available a modern programming language compatible with the latest software technology and is compatible with "best" current programming practice, but the design and implementation of the language should not require additional research or require use of untried ideas. State-of-the-art cannot, however, be taken to mean that a feature has been incorporated in an operational DoD language and used for an extended period, or DoD will be forever tied to the technology of FORTRAN-like languages; but there must be some assurances through analysis and use that its benefits and deficiencies are known. The larger and more complex the structure, the more analysis and use that should be required. Language design should parallel other engineering design efforts in that it is a task of consolidation and not innovation. The language designer should be familiar with the many choices in semantic and syntactic features of language and should strive to compose the best of these into a consistent structure congruous with the needed characteristics. The language should be composed from known semantic features and familiar notations, but the use of proven feature should not necessarily impose that notation. The language must not just be a combination of existing features which satisfy the individual requirements but must be held together by a consistent and uniform structure which acts to minimize the number of concepts, consolidates divergent features and simplifies the whole.

M1. Existing Language Features Only (ALGOL 68)

Satisfied.

The Algol 68 Report was completed in 1968-69. At least one implementation (of nearly the full language) has been operational for several years (at Royal Radar Establishment, England). Several implementations now exist. The features of Algol 68 are within the state of the art. Most of them can be found in a similar form in at least one other implemented language. The revised language is very similar to the original. In particular, certain difficult to translate features have been removed (e.g., proceduring) or corrected (void symbol, separate cast and routine symbols).

M1. Existing Language Features Only (J3B)

Satisfied.

J3B consists of programming constructs which exist either in JOVIAL J3, J73, or other programming languages. The fact that J3B compilers exist and have been used in programing operational military systems means in itself that this requirements is satisfied.

M1. Existing Language Features Only (PASCAL)

Satisfied.

PASCAL was designed in 1969 and has been in practical use since 1970 [Wirth, p. 193]. The language retains the form of its original definition and subsequent changes have been "very few and relatively minor" [p. 133].

M1. Existing Language Features Only (SIMULA 67)

Satisfied.

SIMULA 67 has been implemented on a variety of object machines. Such implementations have existed for several yhears.

M1. Existing Language Features Only (TACPOL)

Satisfied.

TACPOL is an operational language. Consequently all of its features are well within the state of the art.

M2. Unambiguous Definition (TINMAN)

The semantics of the language will be defined unambiguously and clearly. To the extent a formal definition assists in attaining these objectives, the language's semantics will be specified formally.

M2. Unambiguous Definition (ALGOL 68).

Satisfied.

The Algol 68 Revised Report provides a precise, clear (to initiates, at least!), complete, and unambiguous formal definition of the language (both syntax and semantics).

M2. Unambiguous Definition (J3B)

Partially Satisfied.

No formal definition of J3B has been provided, but in practice, when questions have arisen about whether a J3B compiler has correctly implemented the language, it has always been the case that the specification provided a clear answer.

M2. Unambiguous Definition (PASCAL)

Partially Satisfied.

The Revised Report provides a clear, concise definition with only a few gaps (e.g., defining the type matching for parameters and arguments).

There are at least two attempts at a formal language definition:

(1) C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL", ACTA INFORMATICA, 2, 335-355 (1973).

(2) L. Aiello, M. Aiello and R. Weyhrauch, "The Semantics of PASCAL in LCF", Stanford University, Tech. Report CS-74-447, (1974).

M2. Unambiguous Definition (SIMULA 67)

Partially Satisfied.

Palme writes of the "exact definition of SIMULA, which is standardized and works exactly in the same way on all computers with SIMULA systems. One reason for this is that the word "undefined" does not occur in the SIMULA definition as in many other language definitions. Therefore there is very little which can be done differently on different computers" [Palme, p. 38]. The definition of SIMULA 67 [CBL] is indeed an admirably complete document when augmented with the Revised ALGOL 60 Report [AR]. But the language definition is not totally free of ambiguities. For example, the order of argument evaluations for operators is not specified leading to side-effect ambiguities (see C1).

M2. Unambiguous Definition (TACPOL)

Not Satisfied.

The language reference manual is inadequate for precisely defining TACPOL. Examples are used throughout that do not resolve the limitations and the permissible generality of a language feature. Further, a feature is only explained in terms of an accompanying example. Apparent contradictions, omissions, and syntax errors compound the difficulty. A more formal definition was made available to us too late for use in our evaluation of TACPOL; it appears that it may provide a satisfactory definition of TACPOL.

M3. Language Documentation Required (TINMAN)

The user documentation of the language will be complete and will include both a tutorial introductory description and a formal in-depth description. The language will be defined as if it were the machine level language of an abstract digital computer.

The action of any legal program will be determinable from the program and the language description alone. Any computation which can be described in the language will ultimately draw only on capabilities which are built into the language. No characteristics of the source language will be dependent on the idiosyncrasies of its translators.

The language documentation will include syntax, semantics and examples of each language construct, listings of all key words and language defined defaults. Examples shall be included to show the intended use of language features and to illustrate proper use of the language. Particularly expensive and inexpensive constructs will be pointed out. Each document will identify its purpose and prerequisites for its use.

M3. Language Documentation Required (ALGOL 68)

Satisfied.

The Algol 68 Revised Report is the formal definition of the language. C.H. Linsey and S.G. van der Meulen have written an Informal Introduction to Algol 68 (North-Holland, 1973) that gives a very informal tutorial overview initially and then gives a good tutorial treatment of the complete language with lots of examples. The Informal Introduction is currently being revised to correspond to the Revised Report.

M3. Language Documentation Required (J3B)

Partially Satisfied.

An introductory Programmer's Guide exists for J3B, but there is no formal definition of J3B's semantics. Moreover, some aspects of J3B semantics are implementation dependent, e.g., the character code used to implement character strings, and hence, the effect of the BIT operator when applied to character data. These implementation dependent properties of the language are pointed out explicitly in the language specification.

The language specification includes syntax and semantics descriptions, a listing of all key words, and a description of language-defined defaults (such as implicit conversion from integer to fixed in assignment statements). However, no examples of J3B constructs are provided.

The purpose and prerequisites for use of both the language specification and programmer's guide are defined in each document.

M3. Language Documentation Required (PASCAL)

Satisfied.

The current documentation for PASCAL is excellent and is certain to be enriched. It includes:

(1) The two formal definitions cited in M2 above.

(2) The Revised Report by Wirth, which defines the language in clear ALGOL-60 like style.

(3) The User Manual by K. Jensen and N. Wirth, which presents each language feature with examples of use in code fragments and in complete programs. Chapters 0 through 12 cover standard PASCAL, while Chapters 13 and 14 are concerned with PASCAL 6000 specific features.

(4) Systematic Programming by N. Wirth (Prentice-Hall, 1973) is an intelligent introduction to programming using PASCAL as the illustrative language.

M3. Language Documentation Required (SIMULA 67)

Satisfied.

The language definition [CBL] is based on and subsumes part of the revised ALGOL 60 Report [AR]. The book SIMULA BEGIN [B] provides "a tutorial introductory description" with many examples.

M3. Language Documentation Required (TACPOL)

Not Satisfied.

TACPOL's documentation is far from complete. The language reference manual is more in the vein of a tutorial introductory description than a formal in-depth one. Such an in-depth formal description is currently lacking and is sorely needed.

M4. Control Agent Required (TINMAN)

The language will be configuration managed throughout its total life cycle and will be controlled at the DoD level to ensure that there is only one version of the source language and that all translators conform to that standard.

All compilers will be tested and certified for conformity to the standard specification and freedom from known errors prior to their release for use in production projects. The language manager will be on the OSD staff, but a group within the Military Departments or Agencies might act as the executive

agent. A configuration control board will be instituted with user representation and chaired by a member of the OSD staff.

M4. Control Agent Rquired (ALGOL 68)

Not Satisfied.

There is no DoD designated control agent for Algol 68. I.F.I.P. is responsible for the formal definition of the language (the Revised Report), but not for ensuring that implementations satisfy the requirements of the Report.

M4. Control Agent Required (J3B)

Not Satisfied.

There is no DoD designated control authority for J3B.

M4. Control Agent Required (PASCAL)

Not Satisfied.

There is no Dod designated control authority for PASCAL.

M4. Control Agent Required (SIMULA 67)

Not Satisfied.

No DoD SIMULA 67 control agent has been designated. However, there does exist a "SIMULA Standard Group, consisting of representatives for firms and organizations having responsibility for SIMULA 67 compilers" [CBL 1.4]. This group enforces strict standardization and controls future language extensions.

M4. Control Agent Required (TACPOL)

Not Satisfied.

No DoD control agent has been designated, although ARTADS serves as the de facto control agent.

M5. Support Agent Required (TINMAN)

There will be identified support agent(s) responsible for maintaining the translators and for associated design, development, debugging and maintenance aids.

Support of common widely used tools and aids should be provided independent of projects if common software is to be widely used. Support should be on a DoD-wide basis with final responsibility resting with a stable group or groups of qualified in-house personnel.

M5: Support Agent Required (ALGOL 68)

Not Satisfied.

There is no DoD designated support agent for Algol 68.

M5: Support Agent Required (J3B)

Not Satisfied.

There is no DoD agent designated for supporting J3B compilers and programming aids. SofTech, Inc. is currently performing these functions for the Air Force.

M5: Support Agent Required (PASCAL)

Not Satisfied.

There is no DoD agent supporting PASCAL.

M5: Support Agent Required (SIMULA 67)

Not Satisfied.

There is no DoD identified support agent for SIMULA 67.

M5: Support Agent Required (TACPOL)

Not Satisfied.

No DoD support agent has been designated.

M6: Library Standards and Support Required (TINMAN)

There will be standards and support agents for common libraries including application-oriented libraries.

In a given application of a programming language three levels of the system must be learned and used: the base language, the standard library definitions used in that application area, and the local application programs. Users are responsible for the local application programs and local definitions

but not for the language and its libraries which are used by many projects and sites. A principal user might act as agent for an entire application area.

M6: Library Standards and Support Required (ALGOL 68)

Not Satisfied.

There is no DoD designated support agent for Algol 68 application-oriented libraries. Algol 68 does not define a library concept, except that an implementation may supply a library prelude [Section 10.1] to augment the standard prelude declarations [Section 10] and may use pragmats [Section 9.2] to access libraries.

M6: Library Standards and Support Required (J3B)

Not Satisfied.

J3B does not support the concept of libraries of language extensions in the sense intended by the TINMAN (see E1, E3, E4, E5, E6, F4, F5, and F6).

M6. Library Standards and Support Required (PASCAL)

Not Satisfied.

PASCAL does not include the library concept; further, there are no DoD agents supporting PASCAL application-oriented libraries.

M6: Library Standards and Support Required (SIMULA 67)

Although SIMULA 67 is designed to accommodate naturally special application oriented packages [CBL 1-3.4], the library concept is not supported as defined by the TINMAN.

M6: Library Standards and Support Required (TACPOL)

Not Satisfied.

TACPOL does not support the concept of application oriented libraries in the sense intended by M6.

# Candidate Language Evaluation

# and

# Recommendation Report

Performed Under Contract No. F30602-77-C-0009

for

ROME AIR DEVELOPMENT CENTER
GRIFISS AIR FORCE BASE
Rome, New York

International Business Machine Corporation
Federal Systems Division
18100 Frederick Pike
Gaithersburg, Maryland 20760



144 pages (11)

## ABSTRACT

This report contains comparisons between three existing High-Order Languages (HOL) - COBOL, FORTRAN and HAL/S - and the DoD identified HOL requirements for embedded computers as published in the "Requirements for High-Order Computer Programming Languages 'TINMAN'" June 1976. FORTRAN and COBOL are standard DoD High-Order Programming Languages, which address scientific and business programming. HAL/S is a NASA-designed HOL for use on the On-Board Shuttle and other computers. The study was performed for Rome Air Development Center, in support of DoD's HOL cost reduction project through HOL standardization.

The results of this study show that FORTRAN meets relatively few of the requirements for a HOL for embedded computers. COBOL meets a much larger number of the TINMAN requirements and is particularly readable (and hence maintainable) because of its resemblance to English. However, COBOL is not well thought of by the computer-science community because of its wordiness and the inelegance of its program structure. Both FORTRAN and COBOL are widely used languages both by industry and government for non-embedded computer applications, and have been implemented on most general-purpose computers.

HAL/S was designed for programming embedded computers and satisfies an even larger number of the TINMAN requirements than COBOL. The limited number of implementations speaks both for and against HAL/S: it is not widely known and used, but it has a very high degree of standardization.

## ACKNOWLEDGEMENTS

This report was produced in response to Task 4.1.3 in the Statement of Work for the High-Order Language (HOL) effort under Contract Number F30602-77-C-0009. It was delivered to RADC in accordance with Item A002 of the Contract Data Requirements List.

This report was prepared by:

Jean E. Sammet
Maurice Ackroyd
Michael L. Bell
I. Gray Kinnie
Richard S. Kopp

Significant contributions were made by:

Allen R. Mendelin
Thomas Spillman

HAL/S compiler advice was provided by:

Daniel J. Lickly of Intermetrics, Inc.

Contributions were made by the following RADC personnel:

Clement D. Falzarano
Douglas A. White
Samuel A. DiNitto

and the following HOL Working Group personnel:

LTC. William Whitaker of DARPA
Dr. Serafino Amoroso of ECOM
Dr. David Fisher of IDA
Dr. Peter Wegner of Brown University

TABLE OF CONTENTS


| | |
|---|---|
| Title Page | 1 |
| Abstract | 2 |
| Acknowledgements | 3 |
| Table of Contents | 4 |
| Section I - Executive Summary | |
| Ia - Executive Summary Introduction | 5 |
| Ib - Evaluation Matrix | 6 |
| Ic - FORTRAN Language Executive Summary | 9 |
| Id - COBOL Language Executive Summary | 11 |
| Ie - HAL/S Language Executive Summary | 12 |
| Section II - Report Introduction. | 13 |
| Section III - FORTRAN Evaluation | |
| IIIa - FORTRAN Language Introduction | 16 |
| IIIb - FORTRAN Comparative Evaluation | 21 |
| IIIc - FORTRAN Language Features Not Needed | 48 |
| IIId - FORTRAN Summary and Recommendations | 50 |
| Section IV - COBOL Evaluation | |
| IVa - COBOL Language Introduction | 54 |
| IVb - COBOL Comparative Evaluation | 60 |
| IVc - COBOL Language Features Not Needed | 93 |
| IVd - COBOL Summary and Recommendations | 94 |
| Section V - HAL/S Evaluation | |
| Va - HAL/S Language Introduction | 98 |
| Vb - HAL/S Comparative Evaluation | 102 |
| Vc - HAL/S Language Features Not Needed | 134 |
| Vd - HAL/S Summary and Recommendations | 136 |
| Section VI - Comments on TINMAN Requirements | |
| VIa - TINMAN General Comments | 140 |
| VIb - TINMAN Specific Comments | 142 |
| DD1473 | i |

## Section I - EXECUTIVE SUMMARY

### Section Ia - Executive Summary Introduction

The results described in this report were performed under Rome Air Development Center Contract Number F30602-77-C-0009. The effort consisted of investigation and comparison of three existing High-Order Languages (HOL) - COBOL, FORTRAN and HAL/S - to the DoD identified HOL requirements as published in "Requirements for High-Order Computer Programming Languages 'TINMAN'" June 1976. FORTRAN and COBOL are current standard DoD HOL languages. HAL/S is a NASA designed HOL for use in On-Board Shuttle computers. Based upon the comparison IBM-Federal Systems Division has indicated the degree of compliance between the requirements and each of the three languages. Included is an explanation for the compliance ratings, possible changes to bring compliance, and possible eliminations from the three languages. This report has been placed on the ARPANET for use by the DoD HOL Working Group.

The remainder of this executive summary gives a comparison evaluation matrix and three summary recommendations for the languages compared.

A summary of how well each language meets each requirement is presented below. "T" indicates the requirement is almost completely satisified, "P" indicates it is partially satisfied, "F" indicates it fails to satisfy, and "U" (for unknown) indicates it was not possible to determine whether the requirement was satisfied.

| REQUIREMENT | | FORTRAN | COBOL | HAL/S |
|---|---|---|---|---|
| A. | Data and Types | | | |
| A1. | Typed Language | P | T | T |
| A2. | Data Types | P | P | P |
| A3. | Precision | F | P | T |
| A4. | Fixed-Point Numbers | F | T | F |
| A5. | Character Data | F | T | F |
| A6. | Arrays | P | T | T |
| A7. | Records | F | T | T |
| B. | Operations | | | |
| B1. | Assignment and Reference | T | T | T |
| B2. | Equivalence | P | T | T |
| B3. | Relationals | T | T | T |
| B4. | Arithmetic Operations | P | T | P |
| B5. | Truncation and Rounding | P | T | P |
| B6. | Boolean Operations | P | P | T |
| B7. | Scalar Operations - On Arrays | F | T | T |
| B8. | Type Conversion - Implicit | P | F | F |
| B9. | Type Conversion - Explicit | P | P | P |
| B10. | I/O Operations | P | T | P |
| B11. | Power Set Operations | F | F | F |
| C. | Expressions and Parameters | | | |
| C1. | Side Effects | T | T | T |
| C2. | Operand Structure | T | T | T |
| C3. | Expressions Permitted | P | F | T |
| C4. | Constant Expressions | P | F | T |
| C5. | Consistent Parameter Rules | T | P | T |
| C6. | Type Agreement in Parameters | P | F | T |
| C7. | Formal Parameter Kinds | F | F | T |
| C8. | Formal Parameter Specification | F | T | F |
| C9. | Variable Numbers of Parameters | F | F | F |
| D. | Variables, Literals and Constant | | | |
| D1. | Constant Value Identifiers | F | T | T |
| D2. | Numeric Literals | T | T | U |
| D3. | Initial Values of Variables | T | T | T |
| D4. | Numeric Range and Step Size | F | T | F |
| D5. | Variable Types | F | T | P |
| D6. | Pointer Variables | F | F | T |

| | | | | |
|---|---|---|---|---|
| E. | Definition Facilities | | | |
| | E1. User Definitions Possible | P | P | P |
| | E2. Consistent Use of Types | F | F | F |
| | E3. No Default Declarations | F | F | F |
| | E4. Can Extend Existing Operators | F | F | F |
| | E5. Type Definitions | F | F | F |
| | E6. Data Defining Mechanisms | F | F | F |
| | E7. No Free Union or Subset Types | T | T | T |
| | E8. Type Initialization | F | F | F |
| F. | Scope and Libraries | | | |
| | F1. Separate Allocation and Access Allowed | F | F | T |
| | F2. Limiting Access Scope | F | P | T |
| | F3. Compile Time Scope Determination | T | T | T |
| | F4. Libraries Available | P | T | T |
| | F5. Library Contents | F | T | T |
| | F6. Libraries and Compools Indistinguishable | F | T | T |
| | F7. Standard Library Definitions | P | P | T |
| G. | Control Structures | | | |
| | G1. Kinds of Control Structures | P | P | P |
| | G2. The GOTO | T | P | T |
| | G3. Conditional Control | F | P | P |
| | G4. Iterative Control | P | T | P |
| | G5. Routines | F | F | F |
| | G6. Parallel Processing | F | F | T |
| | G7. Exception Handling | F | T | T |
| | G8. Synchronization and Real-Time | F | P | T |
| H. | Syntax and Comment Conventions | | | |
| | H1. General Characteristics | P | P | T |
| | H2. No Syntax Extensions | T | T | T |
| | H3. Source Character Set | T | T | T |
| | H4. Identifiers and Literals | P | P | T |
| | H5. Lexical Units and Lines | F | F | P |
| | H6. Key Words | F | P | T |
| | H7. Comment Conventions | F | P | P |
| | H8. Unmatched Parentheses | T | T | T |
| | H9. Uniform Referent Notation | P | T | T |
| | H10. Consistency of Meaning | F | T | F |
| I. | Default, Conditional Compilation and Language Restrictions | | | |
| | I1. No Defaults in Program Logic | F | T | T |
| | I2. Object Representation Specifications Optional | F | T | F |
| | I3. Compile Time Variables | F | T | T |
| | I4. Conditional Compilation | F | F | P |
| | I5. Simple Base Language | P | F | P |
| | I6. Translator Restrictions | P | F | F |
| | I7. Object Machine Restrictions | P | T | T |

| | | | | |
|---|---|---|---|---|
| J. | Efficient Object Representations and Machine Dependencies | | | |
| | J1. Efficient Object Code | T | P | T |
| | J2. Optimizations Do Not Change Program Effect | P | U | T |
| | J3. Machine Language Insertions | F | P | P |
| | J4. Object Representation Specification | F | T | T |
| | J5. Open and Closed Routine Calls | F | F | P |
| K. | Program Environment | | | |
| | K1. Operating System Not Required | T | T | T |
| | K2. Program Assembly | P | T | T |
| | K3. Software Development Tools | T | P | T |
| | K4. Translator Options | U | P | T |
| | K5. Assertions and Other Optional Specifications | F | P | P |
| L. | Translators | | | |
| | L1. No Superset Implementations | F | F | T |
| | L2. No Subset Implementations | T | F | T |
| | L3. Low-Cost Translation | T | P | U |
| | L4. Many Object Machines | T | T | P |
| | L5. Self-Hosting Not Required | T | T | T |
| | L6. Translator Checking Required | F | U | T |
| | L7. Diagnostic Messages | F | F | F |
| | L8. Translator Internal Structure | T | T | T |
| | L9. Self-Implementable Language | F | T | F |
| M. | Language Definition, Standards and Control | | | |
| | M1. Existing Language Features Only | P | T | T |
| | M2. Unambiguous Definition | F | F | T |
| | M3. Language Documentation Required | F | P | T |
| | M4. Control Agent Required | F | T | T |
| | M5. Support Agent Required | F | U | T |
| | M6. Library Standards and Support Required | F | U | T |
| Totals | T (Fully Satisfies) | 21 | 46 | 59 |
| | P (Partially Satisfies) | 27 | 22 | 18 |
| | F (Fails to Satisfy) | 49 | 26 | 19 |
| | U (Unknown) | 1 | 4 | 2 |

## Section Ic - FORTRAN Language Executive Summary

The FORTRAN language reviewed here is "The American National Standard FORTRAN", ANSI X3.9 - 1966, known here as FORTRAN66. Reference is also made, where appropriate, to FORTRAN76: "The Draft Proposed ANSI FORTRAN" prepared by ANSI Committee X3J3 and published in ACM SIGPLAN Notices as Volume II, No. 3.

FORTRAN is not recommended as a base for the DoD HOL. It is immediately clear, on reading TINMAN, that FORTRAN cannot be modified to satisfy any significant part of the requirements, and still maintain its individuality as a language:

a. FORTRAN's traditional unstructured data specification facilities cannot survive the introduction of composite data structures (A2) particularly when fully generalized (D5). A structured data declaration such as PL/I's DCL is required; DIMENSION, INTEGER, etc., become redundant.

b. Introduction of a fully-partitioned conditional control (G3) makes all of FORTRAN's conditional and switching mechanisms redundant (Logical and Arithmetic IF; Assigned and Computed GOTO). The IF...THEN...ELSE of PL/I requires only minor change.

c. Complete type checking (C6) widens the FORTRAN "domain of compilation" from the traditional "program unit" to the complete executable program. This suggests relaxing the distinction between internally- and externally-defined procedures and leads, reasonably if not inevitably, to the elimination of the familiar FORTRAN function classes: statement functions, intrinsic functions, basic external functions, external functions (these are distinguished mainly by the degree of type checking to be performed by the translator). PL/I satisfies the requirement with only minor change.

d. User specification of allocation and access scope (F1) implies the introduction of storage classes and the elimination of COMMON. PL/I already has storage classes.

e. Reserved keywords (H6), a specific statement delimiter (H1), mnemonic statement identifiers (H1), and the required comments conventions (H7), all require changes to FORTRAN, but none to PL/I.

Thus, by considering only a few requirements, we have eliminated nearly all the distinctive features of FORTRAN - all that distinguishes FORTRAN from a rather weak subset of PL/I. By contrast, PL/I requires very little change to meet those same requirements. In fact, there is probably no positive, as distinct from restrictive, requirement that PL/I does not satisfy at least as well as FORTRAN. The conclusion is evident: PL/I is a far stronger candidate than FORTRAN, which should be eliminated from contention. It is noted that a similar case could be made for ALGOL68, and others, as against FORTRAN. PL/I has been discussed simply because it initially used FORTRAN as a base upon which to add functions to satisfy a large class of problems.

## Section Id - COBOL Language Executive Summary

The language reviewed in this report is the "American National Standard Programming Language COBOL", ANSI X3.23-1974, known herein as COBOL. Reference is also made, where significant differences occur, to the "United States of American National Standard COBOL", ANSI X.23-1968, known as COBOL 68.

Although COBOL is widely used and implemented for its intended class of applications, THERE WOULD PROBABLY BE GREAT PSYCHOLOGICAL RESISTANCE TO ITS ADOPTION AS A BASIS FOR TINMAN WHATEVER COBOL'S TECHNICAL MERITS MIGHT BE WITH RESPECT TO OTHER LANGUAGES.

It is not very surprising to note that COBOL does best in TINMAN requirement areas A, B, D and F. They are, respectively: Data and Types; Operation; Variables, Literals and Constants; and Scopes and Libraries. COBOL is poorest in E (Definition Facilities), since COBOL has none aside from subroutines and does rather poorly in category C (Parameters and Expressions). The entire matter of structure of programs and parameter passage is certainly one of COBOL's weakest concepts.

COBOL 74, considered as a base language for the DoD HOL is:

a. Moderately compliant with existing TINMAN requirements

b. Easily changed to meet approximately 23 of the requirements it does not fully meet

c. Moderately easily changed in approximately 10 of the requirements it does not fully meet

d. Hard or impossible to change in approximately 13 of the requirements it does not fully meet

e. A viable base technically, but probably unacceptable psychologically because of its verbosity and past history, and because language experts have such a low opinion of it.

Note that the numbers above represent complete compliance and include some requirements rated "90%, fully satisfies" in the report.

## Section Ie - HAL/S Language Executive Summary

The HAL/S Language reviewed here is described in the "HAL/S Language Specification" prepared by Neubold and Helmers of Intermetrics, Inc., for The National Aeronautics and Space Administration (NASA). The language was specifically designed for use in programming the flight software of the NASA Space Shuttle Program. HAL/S falls in the class of programming languages whose ancestry includes PL/I, but it differs considerably in features deleted and added.

Considering the total set of TINMAN requirements, HAL/S does not meet DoD specifications for a common language. However, the language could be expanded to provide some of the capabilities required, and thus make HAL/S a base for the design of a common language.

Many of the features required in the TINMAN have not been found necessary for embedded spacecraft computer systems. These include extensibility, generic procedures, and recursion. If these requirements are removed, HAL/S would only fail in ten requirements.

HAL/S, considered as a base language for the DoD HOL:

a. Is compliant with the majority of existing TINMAN requirements

b. Could be easily changed to meet approximately 13 of the requirements it does not fully meet

c. Could be changed with moderate difficulty in approximately 12 of the requirements it does not fully meet

d. Would be very difficult, but not impossible to change in approximately 14 of the remaining requirements it does not fully meet

e. Is a viable base technically, but is probably a poorer choice than some of the larger, more widely used languages.

## Section II - REPORT INTRODUCTION

The effort reflected in this report is in support of the DoD program for software commonality through the adoption of a minimal number of high-order programming languages.

The considerable discussion given to the establishment of a Standard Programming Language for the Department of Defense led to the creation of a Joint Services Committee on January 28, 1975. The purpose of this committee was to determine the language requirements of the services, compare existing languages to these requirements and make the necessary recommendations to DoD.

Currently the Army is using TACPOL, which was developed by Litton, based on PL/I and used to program TACFIRE and TOS. The U.S. Air Force has developed and used certain dialects of JOVIAL, based on ALGOL-58. The U.S. Navy is now using CMS-2. The three services, respectively, have developed specifications for TACPOL II, JOVIAL J73 and CS-4 as their future command and control languages. (CS-4 is based on PASCAL.)

The goal of this effort is to assist DoD in their effort toward software commonality through investigation and comparison of three identified languages to the DoD Higher Order Language (HOL) Working Group's language requirement document "Requirements for High-Order Computer Programming Languages, 'TINMAN'".

IBM-Federal Systems Division has compared the Department of Defense Requirements for High-Order Computer Programming Languages, TINMAN, individually, to ANSI FORTRAN, ANSI COBOL, and HAL/S. Each of the 98 TINMAN requirements was evaluated for degree of compliance and modifications needed. Results of these comparisons were detailed in Section Ib.

For each of the candidate languages, the evaluators determined the degree of compliance to each of the TINMAN requirements and explained how they are met. For each requirement less than fully satisfied, the conflicts were explained, as was the scope of typical modifications necessary to meet the requirement, and the impact of the changes. Other programming considerations and cross conflicts in TINMAN were addressed.

Each of the detailed requirements of TINMAN evaluated against each candidate language used the following criteria and format:

T - Fully satisfies the requirement - meets 90% or greater of the requirement

P - Partially satisfies the requirement - meets 50% to 89% of the requirement

F - Fails to satisfy the requirement - meets less than 50% of the requirement

U - Unknown from the available documents whether the requirement is satisfied or requirement is implementation-dependent (distinction will be made in the comments).

An easy-reference, the Matrix of Compliance, is provided for comparison of the three languages in Section Ib. This provides requirement number, short title of requirement, and the grades T, P, F, or U for each of the three languages.

Section I, the Executive Summary, includes the matrix and an overview of this report. Section II is this Introduction. Sections III, IV and V, respectively, describe the FORTRAN, COBOL and HAL/S evaluations. Each of these sections is organized into an introduction, detailed comparisons to TINMAN requirements to the specific language specifications, a list of unnecessary features and which can probably safely be discarded, and summary and recommendations. Section VI briefly addresses TINMAN as a HOL requirements document.

This report is organized to provide a cohesive look at how the three candidate languages (COBOL, FORTRAN and HAL/S) meet DoD's needs for a HOL for Embedded Computers (TINMAN). At the same time, the report is desiqnd to be parsed into individual language reports (while still remaining cohesive) and eventually split into over 300 individual files on the ARPANET which address individual TINMAN requirements, paragraph by paragraph. Hence, each small group of paragraphs will have a coded heading such as "#.COBOL.Summary and Recommendations", "A. HALS.Data Types", etc.

This report does not reproduce the requirements as stated in TINMAN. Therefore, anyone desiring to read the specific details of the report should first become thoroughly familiar with TINMAN.

The primary references used in this report are:

a. "Requirements for High-Order Computer Programming Languages - 'TINMAN'", June 1976.

b. "American National Standard Programming Language COBOL," ANSI

X3.23-1974, American National Standards Institute, Inc., New York, 1974 (less the Report Writer Module).

c. "United States of American National Standard COBOL," ANSI X3.23-1968, American National Standards Institute, Inc., New York, 1968.

d. "American National Standard (ANS) FORTRAN," ANSI X3.9-1966, American National Standards Institute, Inc., New York, March 1966.

e. "Draft Proposed American National Standard FORTRAN," BSR X-3.9 X3J3/76, American National Standards Committee X3J3, March 1976.

f. Newbold, P. M. and Helmers, C. T., Jr., "HAL/S Language Specification," Intermetrics, Inc., Cambridge, Massachusetts, April 1973.

g. Augmentation Research Center, "NLS Users Guide," Stanford Research Institute, Palo Alto, California, 1976.

Supplemental references include:

a. "CODASYL COBOL Journal of Development (1976)", CODASYL Program Language Committee, Monroeville, Pennsylvania, 1976.

b. Newbold, P. M. and Intermetrics Staff, "HAL/S Programmer's Guide," Intermetrics Inc., including revisions through June 11, 1976.

## Section III - FORTRAN EVALUATION

### Section IIIa - FORTRAN Introduction

#.FORTRAN.Introduction

The FORTRAN language reviewed here is "The American National Standard FORTRAN", ANSI X3.9 - 1966, known here as FORTRAN66. Reference is also made, where appropriate, to FORTRAN76: "The Draft Proposed ANSI FORTRAN", prepared by ANSI Committee X3J3 and published in ACM SIGPLAN Notices as Volume II, No. 3, March 1976. References in the text to FORTRAN refer equally to both FORTRAN66 and FORTRAN76.

FORTRAN76 is a superset of FORTRAN66 (barring minor incompatibilities). Notable additions are:

a. New statements of minor importance (IMPLICIT, INTRINSIC, etc.).

b. Character string data.

c. Arrays may have seven dimensions; upper and lower subscript bounds may be specified.

d. Use of expressions is generalized throughout.

e. A significant amount of new language for I/O.

f. A considerable effort of resolution of semantic ambiguities in FORTRAN66.

Characteristics of the FORTRAN Language

Data Specification

Data specification in FORTRAN66 is unstructured. One may have in one program, for one variable, a type statement:

```
INTEGER*4 X
```

an array declaration:

```
DIMENSION X(10,10)
```

a scope statement:

```
COMMON /C/X
```

an "association" statement:

```
EQUIVALENCE (X(1,1),J)
```

and an initialization statement:

```
DATA X(1,1)/5/
```

These declarations need not appear in juxtaposition and may appear in any order.

Actually, this example is somewhat contrived in that the information in the DIMENSION statement could be carried in the INTEGER or the COMMON statement, and the COMMON and DATA statements are illegal together in any but one kind of program. However, the general concept is clear.

### Execution Structure

With the exception of the DO statement, executable code in FORTRAN is generally unstructured. The most frequently used conditionals in FORTRAN are the "Logical IF" and the "Arithmetic IF", of which the latter may be regarded as a compressed 3-valued CASE statement. The logical IF is not fully partitioned, in fact, it is specifically "one-tailed". Moreover, the "true" branch may contain only one statement, and that neither a DO nor an IF. Thus, nested IFs are not possible and, even when a nest is not required, tortuous formulations like the following are not uncommon:

```
IF A .EQ. B. AND. B.GT.C GOTO 10
GOTO 30
10 DO 20 I = ...
20 CONTINUE
```

Other conditionals in FORTRAN are the "Assigned GOTO" and the "Computed GOTO"; these are somewhat specialized variants of the CASE statement.

The iteration-control DO statement always excepted, the absence of any kind of grouping construct (such as the PL/I BEGIN...END) is particularly notable.

Procedures

Procedures in FORTRAN, except for the MAIN program, are divided into functions and subroutines, differentiated by mode of invocation. Functions are invoked by a functional notation, e.g., A = B + FUNCT (C) and subroutines by a CALL statement CALL SUBR (A,B,C).

The only internal procedure allowed in FORTRAN is the statement function, defined in a single statement almost identical in form to a normal assigned statement. Any other procedure must be defined externally to the program unit in which the reference to the procedure occurs. A program unit is that sequence of FORTRAN statements which defines a MAIN program, a function subprogram, and a subroutine subprogram. It is important to note that throughout FORTRAN it is tacitly assumed that the program unit is also the unit of compilation, and further that the compiler has in general no knowledge of anything outside the program unit. However, in order to facilitate checking of formal and actual parameters, the following classification has arisen:

a. Statement Functions: Type of parameters and result known to the compiler when compiling the program unit which contains the function definition.

b. Intrinsic Functions: A set of predefined functions - the type of parameters and results are always known to the compiler.

c. Basic External Functions: Almost identical to intrinsic functions - the distinction disappears in FORTRAN76 without apparent incompatibility.

d. External Functions: Usually user-defined functions - the type of result may be known if declared in the invoking program. There is no provision for declaring the type of formal parameters in the invoking program.

e. External Subroutines: The type of formal parameters of external subroutines is unknown in the invoking program.

Scope

The normal scope of reference in FORTRAN is the program unit (see c above). The language definitions do not specify nor allow the specification of scope of allocation, which is usually assumed to be coterminous with scope of reference, although this is often untrue in practice.

The exceptions to the general rule (scope of reference equals program unit) are as follows:

a. Scope of reference of formal parameters of a statement function is the statement function definition.

b. Scope of reference of entities contained in named COMMON blocks is all program units within an executable program (see c below) which contain a definition of that named COMMON block.

c. Scope of reference of entities contained in blank COMMON blocks is the executable program. An executable program is a MAIN program together with all directly or indirectly referenced functions and subroutines.

d. A further scope, not contained in any standard language definition, has been added in some implementations. This scope, usually called GLOBAL, is ALL executable programs containing the definition.

Miscellaneous

Immediately apparent characteristics of FORTRAN are the following:

a. Blanks are syntactically non-significant.
b. There is no statement delimiter except end-of-line.
c. Comments may appear only as separate lines.
d. Keywords are not reserved.
e. Statement identifiers are numeric.
f. Both assignment and DO statements use =.

Unreserved keywords pose a non-trivial problem in parsing FORTRAN programs. Also, the combination of this with other factors results in visual near-ambiguities such as the following:

```
DO 30 I = 1.10  is a valid assignment statement;

while DO30I = 1,10  is a valid DO statement.
```

Section IIIb - FORTRAN Comparative Evaluation

The following are paragraph-by-paragraph comparisons of the TINMAN requirements to the candidate language specifications documents:

A1.FORTRAN.Typed Language
Partially Satisfies

FORTRAN is a typed language but types may be acquired by default in FORTRAN66 (5.3).

FORTRAN76 is similar (4.1.2); in addition the IMPLICIT statement (8.5) may be used to specify the default implied type. "Hollerith" data is allowed in literals; there is no corresponding type for variables.

Necessary language modification: eliminate acquisition of type by default; eliminate Hollerith data type.

A2.FORTRAN.Data Types
Partially Satisfies

FORTRAN66 has types for integer (4.2.1), floating-point (4.2.2), and logical (4.2.5). It has no types for fixed-point or character, though FORTRAN76 has a character type (4.8). FORTRAN66 has also double-precision floating-point (4.2.3), complex (4.2.4), and Hollerith (4.2.6; for data only). FORTRAN66 has arrays of up to 3 dimensions (7.2.1.1). FORTRAN76 allows 7 dimensions (5.4.3). FORTRAN has no concept of records as understood in TINMAN; the EQUIVALENCE statement is sometimes used to apprroximate the effect.

Necessary language modifications: elimination of double-precision, complex, and Hollerith types. Addition of fixed-point (see A4) and character (see A5) types. Removal of restrictions on arrays is linguistically simple. Addition of composite data structures (records) together with unrestricted structures (TINMAN:D5) implies a structured data declaration (cf PL/I:DCL) which in turn suggests the elimination of FORTRAN type statements (e.g., INTEGER) and other declaration statements (e.g., DIMENSION). Implementation considerations: a substantial change is necessary to accommodate unrestricted structures. In addition, most implementations are tied to a fixed maximum number of dimensions in arrays: a change would be non-trivial.

A3.FORTRAN.Precision (floating-point)
Fails to Satisfy

FORTRAN66 allows no precision declaration except REAL and DOUBLE PRECISION which are not defined (4.2.3) (except relative to each other). This is also true of FORTRAN76 (4.5). There is no provision for user declaration of the precision of arithmetic.

Necessary language modifications: allow precision declarations for variables and execution scope. Relatively minor syntactically, though the semantics of combinations of precision declarations for variables and scopes would have to be carefully defined. Implementation considerations are relatively minor.

A4.FORTRAN.Fixed-Point Numbers
Fails to Satisfy

FORTRAN has no provision for fixed-point numbers (except integers) as a computational type, though a fixed-point representation of floating-point numbers is allowed on input/output.

Necessary language modifications: definition of fixed-point type including range and step size. Elimination of separate integer type. These are relatively minor, especially as an adjunct to structure definition (A2). Implementation considerations: code generation for all ranges and step-sizes of fixed point to be designed and implemented. Special casing for integers would be required.

A5.FORTRAN.Character Data
Fails to Satisfy

FORTRAN66 has neither character data nor definition by enumeration. FORTRAN76 has character data (4.8) but no definition by enumeration. Moreover, FORTRAN76 has character strings, concatenation, and a substring notation (4.8; 5.7) which are apparently not required by TINMAN and should be eliminated. The collating sequence of character data in FORTRAN76 is not fully defined (3.1.5), apparently to allow both ASCII and EBCDIC to be standard.

Necessary language modifications: definition by enumeration is sufficient; this is considered below (E6). Apparently the authors of TINMAN are content that ordered collections of characters be representable only as arrays. The character data is sufficiently individual to warrant the special notation of strings, substrings, and concatenation.

A6.FORTRAN.Arrays
Partially Satisfies

FORTRAN66 (7.2.1.1; 7.2.1.1.2) requires specification of the number of dimensions, the type, and the subscript range -- the lower bound always being 1. The upper bound of an array which is a parameter to a procedure may be specified either as a further parameter to the procedure or as a constant value within the procedure; however, space for the actual array is allocated at compile time in the invoking procedure. There is no dynamic space allocation; in general, all space may be considered allocated at either compile or load time.

FORTRAN76 (5.1.1.2) allows specification of subscript range as any contiguous sequence of signed integers. There is no provision in either version of FORTRAN for subscript values which are not integers.

Assuming that FORTRAN76 rules to subscript range are adopted, the extension to allow subscript values to be a subsequence from an enumeration type is relatively minor, given definition of types by enumeration as discussed under E6. Space allocation during execution is a concept foreign to FORTRAN, and necessitates the introduction of storage classes - a somewhat radical change (see also F1).

A7.FORTRAN.Records
Fails to Satisfy

FORTRAN has no concept of records (composite data structures).

Given a proper definition of the language changes to allow specification of composite data structures as discussed under A2, no further modification should be necessary to meet this requirement.

B1.FORTRAN.Assignment and Reference
T (Satisfies)

Assignment and reference are defined for all FORTRAN data types. No FORTRAN data type manages its own storage. FORTRAN has no union types.

B2.FORTRAN.Equivalence
Partially Satisfies

The operator .EQ. in FORTRAN66 is defined only in the relational expression X.EQ.Y where X and Y are numeric expressions of the same data type (6.2). In FORTRAN76 X and Y may be numeric of any type (6.3.2), or they may both be character expressions (6.3.4).

Necessary language modifications: extension of the semantics of the EQUIVALENCE operator to cover all data types. Implementation considerations: relatively minor change required.

B3.FORTRAN.Relationals
T (Satisfies)

Relational operators are defined in FORTRAN66 between numeric expressions of the same type (6.2) and in FORTRAN76 between numeric expressions of any type (6.3.2) and between character expressions (6.3.4). FORTRAN has no types defined by enumeration.

Necessary language modifications: definition of data types by enumeration is discussed under E6.

B4.FORTRAN.Arithmetic Operations
Partially Satisfies

The operators +, -, *, / are defined for all FORTRAN data types (6.1). Exponentiation (**) is defined for certain type combinations. FORTRAN66 has no fixed-point data type.

Necessary language modifications: elimination of COMPLEX data type. Addition of fixed-point data type (see A4). Implementation considerations are minor.

B5.FORTRAN.Truncation and Rounding
Partially Satisfies

FORTRAN has no fixed-point data type and no provision for specification of ranges for integer and floating-point data (beyond the undefined relationship between REAL and DOUBLE PRECISION). Thus, the range is implicit in the implementation; however, no high-order truncation is implicit in any language rule.

No language modifications are necessary, except as discussed under A3 and A4.

B6.FORTRAN.Boolean Operations
Partially Satisfies

FORTRAN66 has no "exclusive-or" operator (6.3); neither does

FORTRAN76 (6.4.1). FORTRAN66 allows but does not dictate short circuit evaluation (6.4); similarly FORTRAN76 (6.6.1). However, most implementations perform short circuit evaluation. Note: "NOR" was used in TINMAN when XOR was intended. (Reference 10 November 1976 meeting at IDA)

Addition of the XOR operator is a trivial change. A language rule dictating short circuit evaluation would raise no problems.

B7.FORTRAN.Scalar Operations
Fails to Satisfy

FORTRAN has no array or structure operations; all operations are on array elements.

The addition to the language of array and structure operations is linguistically simple. Compilation of these operations requires, of course, substantial additions to any current implementation.

B8.FORTRAN.Type Conversion
Partially Satisfies

In FORTRAN66, implicit type conversion takes place across assignment (7.1.1.1) but not otherwise (6.1). Explicit conversion operations among numeric data types are provided (Table 3). Most implementations, however, are more permissive and provide implicit conversion in many mixed-type expressions. FORTRAN76 explicitly defines a number of situations in which implicit conversion must take place.

Elimination of implicit type conversion can be effected by simply adopting a language rule to that effect. Diagnosis of violation of the rule would be required of an implementation.

B9.FORTRAN.Changes in Numeric Representation
Partially Satisfies

No explicit conversion is needed between REAL and DOUBLE PRECISION. The greatest part of this requirement concerns fixed-point data, and is discussed, except for the run-time condition, under B5.

Necessary language modifications: add syntax for run-time exception condition. Depending on the model chosen this could have far-reaching consequences since FORTRAN has no provision for such

conditions. Implementation considerations are problematical, but see also B5.

B10.FORTRAN.I/O Operations
Partially Satisfies

FORTRAN66 has the READ and WRITE statements (7.1.3) which interact with symbolic units which may include devices of all kinds. Data may be sent and received, but not control information. FORTRAN76 has considerably more elaborate I/O facilities (12) with provision for passing some kinds of control information. This requirement is not very specific, but it seems certain that the FORTRAN I/O language is both more and less than is required.

Necessary language modifications: uncertain; dependent on elaboration of requirements.

B11.FORTRAN.Power Set Operations
Fails to Satisfy

FORTRAN has no data types defined by enumeration, no data types defined as power sets, and no operations defined on power sets.

Necessary language modifications: given data types defined by enumeration (E6) provision for definition of power sets and of operations on those sets, presents no substantial difficulties. PL/I bit strings might provide the model. Implementation considerations: substantial but not prohibitive addition.

C1.FORTRAN.Side Effects
T (Satisfies)

FORTRAN66 (6.4) outlaws side effects arising from function evaluation; expressions containing integer division must be evaluated left-to-right. FORTRAN76 (6.5.1) similarly outlaws side effects within a statement. Evaluation proceeds left-to-right within a precedence level, except for exponentiation which is right-to-left (i.e., A**B**C means A**(B**C)).

C2.FORTRAN.Operand Structure
T (Satisfies)

Operator hierarchy in FORTRAN is simple and well-defined. (FORTRAN66: 6.1 through 6.4; FORTRAN76: 6.1, 6.4, 6.5)

C3.FORTRAN.Expressions Permitted
Partially Satisfies

FORTRAN66 allows only a restricted set of expressions as subscripts (5.1.3.3); expressions are not allowed as array declarators (7.2.1.1); expressions are not allowed as DO parameters (5.4.2). FORTRAN76 allows unrestricted expressions as subscripts, integer expressions as array declarators (5.1.2), and unrestricted expressions as DO parameters (11.6).

Adoption of FORTRAN76 rules should satisfy this requirement linguistically. No change would be required to many current implementations which already use FORTRAN76 rules.

C4.FORTRAN.Constant Expressions
Partially Satisfies

FORTRAN66 allows only single constants in subscripts (5.1.3.3), as components of array declarators (7.2.1.1), and as non-variable DO parameters (7.1.2.8). FORTRAN76 allows constant expressions in all these cases (5.4.2; 5.1.2; 11.6). The FORTRAN language definitions do not address compile-time evaluation of expressions; this is therefore left to the implementer's discretion.

Adoption of FORTRAN76 rules should satisfy this requirement linguistically. A rule requiring compile-time evaluation could also be adopted though many would consider this not a proper language requirement. No change would be required to many current implementations which already use FORTRAN76 rules, and generally evaluate obvious constant expressions at compile time. There is room for argument as to what constitutes an "obviously constant" expression.

C5.FORTRAN.Consistent Parameter Rules
T (Satisfies)

FORTRAN parameter rules are consistent.

C6.FORTRAN.Type Agreement in Parameters
Partially Satisfies

Both FORTRAN66 (8.1.2; 8.4.2; 8.3.2) and FORTRAN76 (15.4.2; 15.5.2.2; 15.6.2.3) require formal and actual parameters to agree in type, but FORTRAN76 admits one exception: an actual parameter to a subroutine (as distinct from a function) may be a Hollerith constant (8.4.2). Since a formal parameter cannot be defined to be of Hollerith type, no type agreement is possible in this case. The Hollerith data type has been deleted from FORTRAN76 (21.), so this exception no longer exists. However, since rules are given for the extension of FORTRAN76 to include Hollerith data, thereby reinstating the exception (21.7), the distinction between the two language definitions is not profound.

To provide full compliance with the letter of this requirement only elimination of the Hollerith data type is required - obviously a trivial change. The spirit of the requirement, however, requires not only statement of the rule for type agreement but its enforcement. Traditionally, the "domain of compilation" of a FORTRAN translator has been the program unit; that is, a main program, subroutine, or function. Within a program unit, the only procedure which can be both defined and referenced is the statement function. Type checking therefore can be, and routinely is, performed on parameters to statement function invocations. Formal parameters of other procedures invoked within a program unit are, in general, unknown to the translator and thus cannot be checked. Exceptions to this are intrinsic functions and basic external functions (FORTRAN66 terminology). These are language-defined procedures, the characteristics of whose formal parameters are built-in to the translator for the precise purpose of allowing type checking.

Complete type checking can clearly be implemented by widening the domain of compilation to include the complete executable program. No language changes are made absolutely necessary by this, but a number of inherently desirable changes become feasible:

1. Elimination of the distinction between internally- and externally-defined procedures.

2. Elimination of the not very useful statement functions as a special class.

3. Elimination of the classes "intrinsic functions" and "basic external functions".

Complete type checking would be a substantial change to any current implementation.

C7.FORTRAN.Formal Parameter Kinds
Fails to Satisfy

FORTRAN has no concept of parameter classes, though some difficulties with procedure parameters are recognized.

Parameter classes and usage rules must be established. This presents no great syntactic difficulty. Checking during compilation for usage consistent with classification is not trivial.

C8.FORTRAN.Formal Parameter Specifications
Fails to Satisfy

FORTRAN contains no generic procedure capability.

Allowing optional specification of parameter attributes is a minor language change. "Instantiating" a generic procedure at compile time requires non-trivial implementation work.

C9.FORTRAN.Variable Numbers of Parameters
Fails to Satisfy

Procedures with a variable number of arguments cannot be defined in FORTRAN; though certain intrinsic functions with this characteristic are pre-defined and invokable from a FORTRAN program (e.g., MAX, MIN).

Necessary language modifications: allow a variable number of arguments in procedure definitions. There are no serious implementation consideratons.

D1.FORTRAN.Constant Value Identifiers
Fails to Satisfy

Constants cannot be associated with identifiers in FORTRAN66. FORTRAN76 provides the PARAMETER statement for this purpose.

Necessary language modifications: adopt the PARAMETER statement or some other mechanism. The implementation considerations are minimal.

D2.FORTRAN.Numeric Literals
T (Satisfies)

FORTRAN provides a syntax for constants of all built-in data types.

Interpretation of numeric constants is an implementation rather than a language consideration.

No language changes are necessary, and current implementations (as distinct from older implementations) usually provide consistent conversions. Note that assembler conversions are also relevant.

D3.FORTRAN.Initial Values of Variables
T (Satisfies)

Variables, except those in blank COMMON, may be initialized by use of the DATA statement. There are no language-defined default values. The allocation scope is either the named COMMON block or the program unit. Testing for initialization is not a language consideration.

No language changes are necessary, and incomplete compile-time testing for initialization is feasible at reasonable cost; however, run-time testing is potentially very costly.

D4.FORTRAN.Numeric Range and Step Size
Fails to Satisfy

There is no provision in FORTRAN for numeric range specification.

Necessary language modifications: allow range and step-size declarations. These are relatively minor changes given the changes necessary under A2, A4, etc. Implementation costs are variable depending on the degree of run-time checking provided.

D5.FORTRAN.Variable Types
Fails to Satisfy

FORTRAN allows only scalars as array elements, and allows no structures other than arrays.

No syntax changes are required in data specification beyond those for definition of composite data structures (A2), if these are properly defined. There may be a problem devising a tidy notation for referring to, e.g., arrays as elements of an array (perhaps X(I,J(K,L))?).

There is no question that generalizing structures as required here would have a major impact on any current FORTRAN implementation. The

internal tables of FORTRAN compilers tend to be constructed to deal officiently with the restricted attributes of FORTRAN arrays (maximum number of dimensions, scalar elements). The necessary redesign of these tables, in addition to the code generation changes, amounts to a redesign of almost any conceivable FORTRAN compiler.

D6.FORTRAN.Pointer Variables
Fails to Satisfy

FORTRAN has no provision for pointer variables.

Necessary language modifications: relatively simple syntactically (models exist), but with substantial implementation considerations.

E1.FORTRAN.User Definitions Possible
Partially Satisfies

FORTRAN provides no capability for definition of data types and operations, except by definition of functions and subroutines which can be construed as new operations.

A necessary preliminary modification is the provision of a simple uniform grammar (H1). This, and the addition of user-definition facilities, constitute a radical change, amounting to the elimination of most distinguishably FORTRAN characteristics. Implementation considerations are radical. It is doubtful if any current implementation could absorb it.

E2.FORTRAN.Consistent Use of Types
Fails to Satisfy

FORTRAN allows no defined types.

Necessary language modifications are described under E1.

E3.FORTRAN.No Default Declarations
Fails to Satisfy

FORTRAN allows explicit declarations but provides certain defaults.

Necessary language modifications: eliminate default declarations.

E4.FORTRAN.Can Extend Existing Operators
Fails to Satisfy

FORTRAN provides no capability for definition of data types.

Necessary language modifications are discussed under E1.

E5.FORTRAN.Type Definitions
Fails to Satisfy

FORTRAN provides no capability for definition of data types.

The modifications discussed under E1 can easily be changed to meet this requirement.

E6.FORTRAN.Data Defining Mechanism
Fails to Satisfy

FORTRAN provides no capability for definition of data types.

The modifications discussed under E1 can include the required mechanisms without difficulty.

E7.FORTRAN.No Free Union or Subset Types
T (Satisfies)

FORTRAN satisfies this by default since there is no capability for definition of data types.

Modifications discussed under E1 can also be defined in order not to conflict with this requirement.

E8.FORTRAN.Type Initialization
Fails to Satisfy

FORTRAN provides no capability for definition of data types.

Modifications discussed under E1 can be defined to meet this requirement.

F1.FORTRAN.Separate Allocation and Access Allowed
Fails to Satisfy

There is no user facility in FORTRAN for distinguishing between scope of allocation and scope of reference. In FORTRAN66, the distinction is not recognized. In FORTRAN76, the SAVE statement (8.9) allows a distinction to be made in an indirect manner.

The introduction of storage classes or a similar concept is a necessary language modification. This ties in with the modifications discussed under C6 and C7. Depending on the model chosen, the necessary modifications might well be radical.

F2.FORTRAN.Limiting Access Scope
Fails to Satisfy

FORTRAN, with the exception of procedures, has no concept of accessing separately-defined entities.

Necessary language modifications: introduction of explicit name-scoping rules. This would be a radical extension in conjunction with related changes under C6, C7, and F1.

F3.FORTRAN.Compile Time Scope Determination
T (Satisfy)

The scope of all FORTRAN identifiers is wholly determined at compile time. Thus, taken in isolation, this requirement is satisfied by FORTRAN. However, see F1 and F2.

There are no necessary language modifications, but see F1 and F2.

F4.FORTRAN.Libraries Available
Partially Satisfies

A library of procedures is available to FORTRAN users. There is generally no access at compile time. Data definitions are not accessible.

Modifications necessary to meet this requirement form part of those discussed under F5.

F5.FORTRAN.Library Contents
Fails to Satisfy

FORTRAN has no provision for a compile-time accessible library (unless the availability of the interface specifications of intrinsic functions is considered a partial exception).

A facility like the PL/I %INCLUDE would meet this requirement. Implementation of such a facility is not difficult, though non-trivial. Access to, and checking of, interface specifications is part of the work considered under C6 and C7.

F6.FORTRAN.Libraries and Compools Indistinguishable
Fails to Satisfy

FORTRAN provides no compile-time accessible library. However, given the change discussed under F5, there seems no reason to distinguish libraries and compools.

No language changes are necessary beyond those mentioned under F5.

F7.FORTRAN.Standard Library Definitions
Partially Satisfies

FORTRAN66 provides standard machine-independent interfaces to input/output devices. These would probably not be regarded as adequate by the authors of TINMAN, since the nature of the interfaces is largely dictated by first-generation ranking. FORTRAN76 provides more up-to-date interfaces, though still inadequate. Some implementations provide subroutine interfaces to hardware clocks, etc., though these cannot be considered standard.

Specification of language would necessitate careful consideration of the full range of devices, special hardware, and other external facilities to be supported. Implementation to the defined interfaces should not be difficult.

G1.FORTRAN.Kinds of Control Structures
Partially Satisfies

FORTRAN provides mechanisms for sequential, conditional and iterative control. It provides none for recursion, parallel processing, exception handling and asynchronous interrupt handling.

Necessary language modifications: see later requirements in this section (G3 through G8).

G2.FORTRAN.The GOTO
T (Satisfies)

The destination of the FORTRAN GOTO is limited to the program unit, which is the most narrow scope defined in FORTRAN (with the exception of the statement function, which cannot contain a GOTO).

G3.FORTRAN.Conditional Control
Fails to Satisfy

FORTRAN provides no fully-partitioned conditional control. The "arithmetic IF" (FORTRAN66: 7.1.2.2) is more properly a 3-valued CASE statement; the "logical IF" (FORTRAN66: 7.1.2.3) is specifically "one-tailed" (no ELSE branch). Moreover, only one statement (and that neither a DO nor a logical IF) may be executed on the "true" branch. Thus, nesting of conditionals is impossible. The "Assigned GOTO" is another variant on a CASE statement (FORTRAN66: 7.1.2.1.2): a numeric statement identifier is "assigned" to an integer variable whose value is tested at the decision point. The "Computed GOTO" is a more orthodox CASE statement (FORTRAN66: 7.1.2.1.3): the value of an integer variable is used as an index to a list of statement identifiers.

Necessary language modifications: provide IF...THEN...ELSE as described in TINMAN; eliminate Logical and Arithmetic IF; eliminate Assigned GOTO; and provide grouping control (e.g., BEGIN...END).

G4.FORTRAN.Iterative Control
Partially Satisfies

The FORTRAN DO permits entry only at the head; but the DO-variable is not local to the loop, and termination is not allowed in the middle of the loop. FORTRAN66 evaluates the termination condition at the end of the loop, while FORTRAN76 evaluates it at the head of the loop (see FORTRAN76: 20.11).

Necessary language modifications: redesign loop control as required.

G5.FORTRAN.Routines
Fails to Satisfy

FORTRAN has no provision for the definition of recursive routines.

Necessary language modifications: though the provision of the ability to define recursive routines is syntactically not difficult, the implications are radical in the context of FORTRAN, and amount to a redefinition of the language. See C6, C7, F1, F2.

G6.FORTRAN.Parallel Processing
Fails to Satisfy

FORTRAN has no provision for parallel processing. The subroutine package devised by the Instrumentation Society of America (ISA) and augmented variously by implementers provides a partial solution.

Necessary language modifications: provide task identifiers, scheduling mechanisms, etc. See also G8. The underlying mechanisms required for implementation is a substantial piece of work for each object machine.

G7.FORTRAN.Exception Processing
Fails to Satisfy

FORTRAN provides no mechanism for exception handling, except that certain well-defined input/output exceptions are provided for in FORTRAN76 (12).

Necessary language modifications: provide a modified PL/I ON-unit capability or its equivalant. These are not trivial changes.

G8.FORTRAN.Synchronization and Real-Time
Fails to Satisfy

FORTRAN has no provision for real-time; the ISA subroutine package provides some of the required facilities (see G6).

Language modifications required: in addition to modifications in G6, provide the ability to delay tasks, specify priorities, accept asynchronous interrupts, and access real-time clocks. Relatively sophisticated control must be provided for each object machine.

H1.FORTRAN.General Characteristics
Partially Satisfies

FORTRAN66 is almost free format (except that columns 1-6 of the input line are reserved for statement identifiers, comments and continuation codes). It has no explicit statement delimiter, allows mnemonically significant (but short) names for data, but only numeric statement identifiers; has a simple, easily parsed, but not uniform grammar; has no unique notations (depending on the definition), allows no abbreviations, and admits to no syntactic ambiguities.

Language modifications required: eliminate special significance of columns 1-6; add explicit statement delimiter; eliminate grammatical non-uniformities (a substantial list); allow non-numeric identifiers of arbitrary length for both statements and variables; and make keywords reserved to simplify parsing. Note that these changes, though apparently simple, would in themselves make the resulting language barely recognizable as FORTRAN. Implementation considerations are difficult to assess in isolation. The effects of these changes and those discussed elsewhere are enough to transform the language, and thus the implementation, entirely.

H2.FORTRAN.No Syntax Extensions
T (Satisfies)

No syntax extensions are permitted in FORTRAN.

H3.FORTRAN.Source Character Set
T (Satisfies)

Any FORTRAN66 program may be written using only 47 characters (which are all contained in the ASCII subset) (3.1). FORTRAN76 (3.1.4) adds the apostrophe and the colon for a total of 49.

H4.FORTRAN.Identifiers and Literals
Partially Satisfies

FORTRAN66 provides formation rules for identifiers and numeric literals (3.4; 3.5; 5.1.1) and FORTRAN76 rules are identical. No break character is specified. Blanks or spaces are ignored and may be used for clarity. FORTRAN66 gives rules for Hollerith constants (5.1.1.6). FORTRAN76 gives rules for character literals: there is no break character and spaces are significant. Separate quoting is neither required nor allowed; long literals (Hollerith or character) must be continued at the beginning of the next line.

Language modifications required: provide break character for identifiers and literals and possibly eliminate non-significance of

blanks. Provide a separate-quoting convention and require its use for long literals by disallowing continuation of literals over end-of-line.

H5.FORTRAN.Lexical Units and Lines
Fails to Satisfy

Lexical units in FORTRAN may be continued across lines at will, with the exception of the END statement (occurs once in a program unit) which must be contained on a single line. Object characters are not defined in the FORTRAN language; the practical effect is that they may in most cases safely be included in literal strings, but there is no guarantee that such usage may not conflict with the internal conventions of a particular translator.

Adoption of a rule against continuing lexical units across lines is easy, but enforcement of the rule would be a burden for a translator without further changes. In particular, the non-significance of blanks, unreserved keywords, and the default definition of data names, make impossible the immediate recognition of a lexical unit by a translator. Elimination of these factors (which except for the first is explicitly required elsewhere in TINMAN), would ease the burden considerably. In conjunction with this rule, definition of a literal string as an instance of a lexical unit which must not be continued across end-of-line (as required by H4) would seem to be sufficient to allow safe inclusion of object characters in the string.

H6.FORTRAN.Key Words
Fails to Satisfy

FORTRAN key words are not reserved. By rough count there are 32 keywords in FORTRAN66; this does not include logical (3) and relational (6) operators, FORMAT field descriptors (9), and names of intrinsic and basic external functions (54). It also excludes arithmetic operators and other basic syntactical symbols (parentheses, comma, slash, =).

Reserving key words is simple and in accordance with good FORTRAN programming practice. The benefit would be felt by the implementers, who could eliminate the absurd double-parsing necessary to recognize that the statement "DO 30 I = 1.10" is a perfectly legitimate assignment statement by existing FORTRAN rules. The statement "DO30I = 1,10" is equally a legitimate DO statement. Here again as in H5, the non-significance of blanks must be abolished to realize the full intent of reserving key words.

H7.FORTRAN.Comment Conventions
Fails to Satisfy

Comments may appear only on separate lines. They are introduced by the character C in column 1 of such a line in FORTRAN66 (3.2.1). In FORTRAN76 the character * may be substituted (3.2.1).

Adoption of a more flexible comment convention, such as the PL/I /*...*/ pair, should introduce no difficulties.

H8.FORTRAN.Unmatched Parentheses
T (Satisfies)

FORTRAN formally allows no unmatched parentheses and so satisfies this requirement. However, use of parentheses in FORTRAN is strictly local - a left parenthesis must be matched by a right parenthesis within the same statement. Parenthetical constructs in the larger and more important sense - such as the BEGIN...END pair - do not exist in FORTRAN, although some such construct would have to be introduced if FORTRAN were to be modified to meet TINMAN requirements.

H9.FORTRAN.Uniform Referent Notation
Partially Satisfies

There are restrictions on subscript forms in FORTRAN66. Elsewhere, TINMAN requires the elimination of such restrictions and they are eliminated in FORTRAN76. Given that elimination, there is no syntactical distinction in FORTRAN between a function reference and an array reference, so the requirement is satisfied with respect to arrays. (No data structures other than arrays are defined in FORTRAN.) As to scalar data, technically the question does not arise in FORTRAN66 which does not recognize the possibility of parameterless functions, but requires an invocation of such a function to include an empty parenthesis as a parameter list (15.5.1, 15.2.1). Hence, a reference to scalar data may not be replaced without change by a reference to a parameterless function.

Given that restrictions on subscript forms are eliminated, elimination of empty parameter lists is all that is required to make FORTRAN satisfy this requirement with respect to existing data structures). Since other composite data structures are also a rquirement (A2), the referent notations for such structures must be defined appropriately.

H10.FORTRAN.Consistency of Meaning
Fails to Satisfy

The operator '=' functions both as an assignment operator and in

the specification of DO parameters. A statement function definition (FORTRAN66: 8.1.1) is syntactically indistinguishable from a normal assignment statement.

Language modifications required: respecify iteration control and eliminate statement function.

I1.FORTRAN.No Defaults in Program Logic
Fails to Satisfy

The following is a selection of instances of defaults from FORTRAN66 (there are many others): (1) action on encountering end-of-file on READ is undefined (7.1.3.3.3), (2) action at Computed GOTO defined only for values within range (7.1.2.1.3), (3) loop control variable is undefined on loop exit (7.1.2.8.1), and (4) precision of floating-point data is implementer-defined (4.2.3). FORTRAN76 corrects the first three cases (12.7; 11.2; 11.6.7) and many others. However, it does not define precision of floating point, and only partially specifies the collating sequence of character data.

Adoption of FORTRAN76 rules would correct many of the defects in FORTRAN66. Correction of the explicit FORTRAN76 omissions would further improve the language. However, without painstaking study there is no way of knowing how many implicit defaults still remain.

I2.FORTRAN.Object Representation Specification Optional
Fails to Satisfy

FORTRAN66 does not specify object representations which are therefore implementer-defined. There is no override capability. FORTRAN76 defines the relative sizes of storage units for defined data types (17.1.1) and defines the ordering of arrays (5.4.2) but is otherwise silent on object representation. Again, there is no override capability.

Language modifications required: specify default object representation and provide override capability.

I3.FORTRAN.Compile Time Variables
Fails to Satisfy

There is no provision for environmental inquiry in FORTRAN, though a number of implementations have allowed user specification. However, the action taken was usually implementer-defined.

Language modifications required: provide for interrogation of compile-time variables.

I4.FORTRAN.Conditional Compilation
Fails to Satisfy

FORTRAN does not define any conditional compilation capability.

Language modifications required: provide conditional compilation capability.

I5.FORTRAN.Simple Base Language
Partially Satisfies

FORTRAN66 is a simple language, however, with some redundancies; e.g., (1) Computed GOTO and Assigned GOTO, (2) logical and arithmetic IF, and (3) ability to specify arrays in either DIMENSION or type statements.

FORTRAN, even with redundancies removed, seems hardly suitable as a base for extension.

I6.FORTRAN.Translator Restrictions
Partially Satisfies

FORTRAN66 specifies the size of identifiers and the maximum number of array dimensions, but does not specify number of identifiers or other factors relevant to translator design, which therefore becomes implementer-defined.

Language modifications required: specify suitable translator requirements.

I7.FORTRAN.Object Machine Restrictions
Partially Satisfies

FORTRAN defines no object machine restrictions. An implementation, however, is not prevented from doing so.

Language modifications required: declare arbitrary restrictions illegal.

J1.FORTRAN.Efficient Object Code
T (Satisfies)

FORTRAN, of course, can be very efficiently implemented. However, the language as modified to meet a substantial subset of the TINMAN requirements would present a more difficult problem.

J2.FORTRAN.Safe and Effective Optimization Possible
Partially Satisfies

FORTRAN is simple enough to allow safe optimization in most cases. Some problems remain. For instance, although it is forbidden for a statement to contain a function reference which has side effects upon the containing statement, there is no way of ensuring that this is the case.

Language modifications required: define precise conditions which must be met before optimization is allowed.

J3.FORTRAN.Machine Language Insertions
Fails to Satisfy

FORTRAN defines no mechanism for machine-language insertions.

Language modifications required: define a machine-language insertion mechanism.

J4.FORTRAN.Object Representation Specification
Fails to Satisfy

FORTRAN does not support composite data structures or the specification of their object representation.

Language modifications required: support for composite data structures as under A2 and allowance for specification of object representation.

J5.FORTRAN.Open and Closed Routine Calls
Fails to Satisfy

FORTRAN does not define a method of specifying whether calls are

open or closed. While no rule is laid down, the assumption is that called routines are externally defined and are not available for in-line inclusion.

Language modifications required: allow for user specification of open or closed calls.

K1.FORTRAN.Operating System Not Required
T (Satisfies)

FORTRAN requires no operating system. Certain features such as input/output, do require run-time support routines. It should be noted that perhaps the full range of TINMAN facilities -- parallel processing, task scheduling, etc. -- requires an array of run-time services which may be hard to distinguish formally from an operating system.

K2.FORTRAN.Program Assembly
Partially Satisfies

Separate definition and separate compilation is the assumed (though not required) method of operation in FORTRAN. It is assumed that the integration of separately compiled modules will be performed by a separate mechanism (e.g., link editor) rather than under the control of the translator.

Language modifications required: the integration of modules by the translator is a minor variant on the "domain of compilation" changes for type-checking, library support, etc. discussed under C6, C7, F1, F2.

K3.FORTRAN.Software Development Tools
T (Satisfies)

This is not entirely a language requirement; the FORTRAN language definition does not address software development tools. However, as a widely used scientific programming language, FORTRAN has many tools available.

K4.FORTRAN.Translator Options
Unknown

This is not a language requirement. The FORTRAN language definition does not address translator options.

K5.FORTRAN.Assertions and Other Optional Specifications
Fails to Satisfy

FORTRAN makes no provision for optionally interpretable comments.

Definition of a method of denoting optionally interpretable comments would be trivial.

L1.FORTRAN.No Superset Implementations
Fails to Satisfy

FORTRAN66 specifically allows superset implementations (Appendix B1) as does FORTRAN76 (1.3).

Modify specifications to forbid superset implementations.

L2.FORTRAN.No Subset Implementations
T (Satisfies)

Subset implementations of FORTRAN66 do not conform to the standard (Appendix B1). A subset language is defined in FORTRAN76. Translators conform to the subset standard if, at the very least, the subset language is provided (1.3.1).

L3.FORTRAN.Low-Cost Translation
T (Satisfies)

Many implementations bear witness that low-cost translation of FORTRAN is possible. However, translation of FORTRAN as modified to meet a substantial number of the TINMAN requirements presents a different problem. Note: As an implementation requirement, this is in partial conflict with L4.

L4.FORTRAN.Many Object Machines
T (Satisfies)

Not a language requirement; however, FORTRAN probably has more implementations than any other language. Note: For one translator to produce code for a variety of object machines, a machine-independent intermediate language is required. Design of such a language is a problem almost as formidable as design of a programming language. Further, this precludes one-pass compilation, which is one way of

meeting requirement L3. More subtly, the necessary generality of an intermediate language may obscure correspondences between the original input and the object machine with the effect that more work is done for potentially less efficient results.

L5.FORTRAN.Self-Hosting Not Required
T (Satisfies)

FORTRAN translators have been written for a wide variety of host machines, some of them quite small. Note: It is unlikely that the same design would be used for both a translator for a small-host machine, and a translator for many object machines (L4).

L6.FORTRAN.Translator Checking Required
Fails to Satisfy

This is not a language requirement (but see C6). (No language modifications required.)

As discussed under C6, checking of formal against actual parameters necessitates a departure from the traditional FORTRAN concept of the program unit as the "domain of compilation".

L7.FORTRAN.Diagnostic Messages
Fails to Satisfy

The FORTRAN language definition does not prescribe diagnostic messages or translator actions for erroneous programs. FORTRAN66 mentions the possibility of standardizing "diagnostics" but dismisses it as "premature". FORTRAN76 does not discuss it.

Prescribe diagnostic messages and translator action. Note that, as mentioned in FORTRAN 66, prescription of diagnostic messages and translator action may have profound implications for translator design. (Reference Appendix B1)

L8.FORTRAN.Translator Internal Structure
T (Satisfies)

Neither FORTRAN66 nor FORTRAN76 dictates translator characteristics.

L9.FORTRAN.Self-Implementable Language
Fails to Satisfy

Translators have been written in FORTRAN, but only with some special extensions made to the language. As defined, however, FORTRAN is not a suitable language for writing language translators.

If modifications were made to FORTRAN to meet any reasonable number of TINMAN requirements, particularly A2 (Composite Data Structures), the resulting language would be suitable for self-implementation.

M1.FORTRAN.Existing Language Features Only
Partially Satisfies

FORTRAN76 is well within the state of the art, as is, even more so, FORTRAN66. The question remains whether modifications to meet TINMAN requirements could be made within the state of the art. There seems no reason why this could not be accomplished.

M2.FORTRAN.Unambiguous Definition
Fails to Satisfy

There is no formal definition of FORTRAN. A good part of FORTRAN76 is evidence of the semantic ambiguities of FORTRAN66. In this respect, FORTRAN76 might be considered the unambiguous definition of FORTRAN66, though it is not clear how many ambiguities remain. However, it should be pointed out that a formal definition gives only the comforting appearance of infallibility. In reality, it needs to be syntactically and semantically debugged as carefully as any other program.

No language modifications; develop formal definition.

M3.FORTRAN.Language Documentation Required
Fails to Satisfy

Innumerable descriptions exist of the FORTRAN language and of particular implementations. It is doubtful whether any meets the requirement as stated.

Provide language documentation as specified.

M4.FORTRAN.Control Agent Required

Fails to Satisfy

This is not a language requirement. DoD is free to control (within its own domain) any language it adopts. As it stands, FORTRAN is controlled by ANSI, and thus there would be difficulties arising from divergent interpretations.

Set up strongly supported compiler validation agency.

M5.FORTRAN.Support Agent Required
Fails to Satisfy

This is not a language requirement.

No language modifications are required. Presumably the validation agency set up under M4 could act as the support agency as well as manage configuration for compilers, tools, libraries and standards.

M6.FORTRAN.Library Standards and Support Required
Fails to Satisfy

This is not a language requirement. FORTRAN does not support libraries in the sense intended by TINMAN (see E1, F4, F5).

Presumably the validation agency set up under M4 could act as the support agency as well as manage configuration for compilers, tools, libraries and standards.

Section IIIc - FORTRAN Language Features Not Needed

#.FORTRAN.Deletions From FORTRAN66 To Meet TINMAN Requirements

As noted elsewhere, modification of FORTRAN to meet TINMAN requirements must result in a major transformation of the language. It is thus difficult to distinguish in all cases between language which should be deleted because its function is not required, and language which should be deleted because it is inadequate to represent the required function. An attempt can however be made:

1. The (admittedly trivial) functions represented by the PAUSE and STOP statements are not required. These statements can therefore be deleted.

2. Under TINMAN I5 ("Simple Base Language"), language features must provide a simple unduplicated capability. Under this rule the COMPLEX data type, together with its associated operations, "intrinsic" functions, and "basic external" functions (CONJG, CSORT, etc.) must be eliminated: COMPLEX can clearly be represented as a "defined type". Hollerith data is a duplication of the required character data (A2) and should be eliminated. The function of the FORMAT statement is largely an implicit duplication of the required explicit object-to-character conversion (B8): the FORMAT statement should be eliminated. Either the arithmetic or logical IF should be eliminated as a duplicate function; however, neither is adequate to meet the requirements (see below). A precisely similar argument applies to the Computed and Assigned GOTO.

3. The bulk of the eliminations arise from the inadequacy of the FORTRAN language features, as defined. Typically, TINMAN calls for a general capability while FORTRAN traditionally has provided a more restricted capability. Assuming that the general capability is provided, the FORTRAN feature becomes redundant and should be eliminated. These questions are discussed under the particular requirements and may not be strictly relevant to this section; however, it seems appropriate to provide a consolidated list:

| Requirement | FORTRAN Eliminated | Comments |
|---|---|---|
| A2, A7, D5 | Type statements<br>DIMENSION<br>EQUIVALENCE<br>DATA | Unrestricted structures require structured data declaration. |
| A3 | Double-precision data type | Precision specifications rule out arbitrary (machine-dependent) precision. |
| C6 | Statement functions.<br>Intrinsic function.<br>Basic external functions. | Universal type checking suggests elimination distinctions between internal and external function definitions. |
| F1 | COMMON | General mechanism to specify allocation and reference scope of variables rules out the restricted FORTRAN solution |
| G3 | Arithmetic IF<br>Logical IF<br>Computed GOTO<br>Assigned GOTO | These features are inadequate to meet the requirement for fully-partitioned conditionals. |
| G4 | DO | The FORTRAN DO does not meet the stated requirement for an iterative control structure. |
| H1 | Numeric statement identifiers | Identifiers are required to be mnemonic. |
| H7 | Comments convention | Comments must be able to appear "anywhere reasonable". A separate line is not sufficient. |

## Section IIId - FORTRAN Summary and Recommendations

The following is a table showing the TINMAN requirements by category and the requirements which FORTRAN fully or partially satisfy or failed to satisfy as well as those which are undetermined (unknown).

| REQUIREMENT | | FORTRAN |
|---|---|---|
| A. | Data and Types | |
| | A1. Typed Language | P |
| | A2. Data Types | P |
| | A3. Precision | F |
| | A4. Fixed-Point Numbers | F |
| | A5. Character Data | F |
| | A6. Arrays | P |
| | A7. Records | F |
| B. | Operations | |
| | B1. Assignment and Reference | T |
| | B2. Equivalence | P |
| | B3. Relationals | T |
| | B4. Arithmetic Operations | P |
| | B5. Truncation and Rounding | P |
| | B6. Boolean Operations | P |
| | B7. Scalar Operations - On arrays | F |
| | B8. Type Conversion - Implicit | P |
| | B9. Type Conversion - Explicit | P |
| | B10. I/O Operations | P |
| | B11. Power Set Operations | F |
| C. | Expressions and Parameters | |
| | C1. Side Effects | T |
| | C2. Operand Structure | T |
| | C3. Expressions Permitted | P |
| | C4. Constant Expressions | P |
| | C5. Consistent Parameter Rules | T |
| | C6. Type Agreement in Parameters | P |
| | C7. Formal Parameter Kinds | F |
| | C8. Formal Parameter Specification | F |
| | C9. Variable Numbers of Parameters | F |
| D. | Variables, Literals and Constant | |
| | D1. Constant Value Identifiers | F |
| | D2. Numeric Literals | T |
| | D3. Initial Values of Variables | T |
| | D4. Numeric Range and Step Size | F |
| | D5. Variable Types | F |
| | D6. Pointer Variables | F |
| E. | Definition Facilities | |

| | | |
|---|---|---|
| E1. | User Definitions Possible | P |
| E2. | Consistent Use of Types | F |
| E3. | No Default Declarations | F |
| E4. | Can Extend Existing Operators | F |
| E5. | Type Definitions | F |
| E6. | Data Defining Mechanisms | F |
| E7. | No Free Union or Subset Types | T |
| E8. | Type Initialization | F |
| F. | Scope and Libraries | |
| F1. | Separate Allocation and Access Allowed | F |
| F2. | Limiting Access Scope | F |
| F3. | Compile Time Scope Determination | T |
| F4. | Libraries Available | P |
| F5. | Library Contents | F |
| F6. | Libraries and Compools Indistinguishable | F |
| F7. | Standard Library Definitions | P |
| G. | Control Structures | |
| G1. | Kinds of Control Structures | P |
| G2. | The GOTO | T |
| G3. | Conditional Control | F |
| G4. | Iterative Control | P |
| G5. | Routines | F |
| G6. | Parallel Processing | F |
| G7. | Exception Handling | F |
| G8. | Synchronization and Real-Time | F |
| H. | Syntax and Comment Conventions | |
| H1. | General Characteristics | P |
| H2. | No Syntax Extensions | T |
| H3. | Source Character Set | T |
| H4. | Identifiers and Literals | P |
| H5. | Lexical Units and Lines | F |
| H6. | Key Words | F |
| H7. | Comment Conventions | F |
| H8. | Unmatched Parentheses | T |
| H9. | Uniform Referent Notation | P |
| H10. | Consistency of Meaning | F |
| I. | Default, Conditional Compilation and Language Restrictions | |
| I1. | No Defaults in Program Logic | F |
| I2. | Object Representation Specifications Optional | F |
| I3. | Compile Time Variables | F |
| I4. | Conditional Compilation | F |
| I5. | Simple Base Language | P |
| I6. | Translator Restrictions | P |
| I7. | Object Machine Restrictions | P |
| J. | Efficient Object Representations and Machine Dependencies | |

| | | |
|---|---|---|
| J1. | Efficient Object Code | T |
| J2. | Optimizations Do Not Change Program Effect | P |
| J3. | Machine Language Insertions | F |
| J4. | Object Representation Specification | F |
| J5. | Open and Closed Routine Calls | F |
| K. | Program Environment | |
| K1. | Operating System Not Required | T |
| K2. | Program Assembly | P |
| K3. | Software Development Tools | T |
| K4. | Translator Options | U |
| K5. | Assertions and Other Optional Specifications | F |
| L. | Translators | |
| L1. | No Superset Implementations | F |
| L2. | No Subset Implementations | T |
| L3. | Low-Cost Translation | T |
| L4. | Many Object Machines | T |
| L5. | Self-Hosting Not Required | T |
| L6. | Translator Checking Required | F |
| L7. | Diagnostic Messages | F |
| L8. | Translator Internal Structure | T |
| L9. | Self-Implementable Language | F |
| M. | Language Definition, Standards and Control | |
| M1. | Existing Language Features Only | P |
| M2. | Unambiguous Definition | F |
| M3. | Language Documentation Required | F |
| M4. | Control Agent Required | F |
| M5. | Support Agent Required | F |
| M6. | Library Standards and Support Required | F |

| | | |
|---|---|---|
| Totals | T (Fully Satisfies) | 21 |
| | P (Partially Satisfies) | 27 |
| | F (Fails to Satisfy) | 49 |
| | U (Unknown) | 1 |

Recommendation

FORTRAN is not recommended as a base for the DoD HOL. To the reader of TINMAN, it is clear that FORTRAN cannot be modified to satisfy any significant part of the requirements, while maintaining its individuality as a language:

1. FORTRAN's traditional unstructured data specification facilities cannot survive the introduction of composite data structures (A2) especially when fully generalized (D5). A

structured data declaration such as PL/I's DCL is required; DIMENSION, INTEGER, etc. become redundant.

2. Introduction of a fully partitioned conditional control (G3) effectively makes redundant all of FORTRAN's conditional and switching mechanisms (Logical and arithmetic IF; Assigned and Computed GOTO). The IF...THEN...ELSE of PL/I requires only minor change.

3. Complete type checking (C6) widens the FORTRAN "domain of compilation" from the traditional "program unit" to the complete executable program. This suggests relaxing the distinction between internally and externally defined procedures and leads, logically, to the elimination of the familiar FORTRAN function classes: statement functions, intrinsic functions, basic external functions, external functions (which are distinguished from each other mainly by the degree of type checking to be performed by the translator). PL/I satisfies the requirement with only minor change.

4. User specification of allocation and access scope (F1) implies the introduction of storage classes and the elimination of COMMON. PL/I already has storage classes.

5. Reserved keywords (H6), a specific statement delimiter (H1), mnemonic statement identifiers (H1) and the required comments conventions (H7) all require changes to FORTRAN, but none to PL/I.

Thus, by consideration of only a few requirements, we have eliminated nearly all the distinctive features of FORTRAN - all that distinguishes FORTRAN from a rather weak subset of PL/I. By contrast, PL/I requires very little change to meet those same requirements. In fact, there is probably no positive (as distinct from restrictive) requirement that PL/I does not satisfy at least as well as FORTRAN. The conclusion is inescapable: PL/I is a far stronger candidate than FORTRAN, which should be eliminated from contention.

It is understood, of course, that a similar case could be made for ALGOL68, and others, as against FORTRAN.

PL/I has been discussed simply because it used FORTRAN as a base upon which to add functions to satisfy a large class of problems.

## Section IV - COBOL EVALUATION

### Section IVa - COBOL Language Introduction

#.COBOL.Introduction

The primary evaluation of COBOL with respect to individual TINMAN requirements has been made using ANSI Standard COBOL 74; thus, wherever the word "COBOL" is used it should be understood that it means (only) the ANSI Standard COBOL 74. COBOL 74 is defined by the "American National Standard Language COBOL", ANSI X3.23-1974, American National Standards Institute, New York, NY, 1974.

The evaluation of individual TINMAN requirements with regard to COBOL 68 was done after the COBOL 74 evaluations. There are very few changes in the individual evaluations of Fully, Partially, or Fail due to differences between COBOL 68 and 74. For that reason, no comments about COBOL 68 have been made unless the actual evaluation for a specific requirement has been changed due to differences between COBOL 68 and 74. However, it is important to note that some of the text discussion, and specific illustrations from the language, and the rationale behind the COBOL 74 evaluation does not (necessarily) apply to COBOL 68; only the final evaluation is the same if no mention of COBOL 68 is made. COBOL 68 is defined by the "United States of America National Standard COBOL", ANSI X.23-1968, American National Standards Institute, New York, NY, 1968.

When appropriate, separate paragraphs are provided which indicate the level of difficulty in changing COBOL 74 to satisfy the TINMAN requirements. In considering this aspect, some attention was paid to the "CODASYL 1976 Journal of Development" which represents the current definition of COBOL, although it is not a standard.

FUNDAMENTAL CONCEPTS OF COBOL

Four Divisions:

The most fundamental concept of COBOL is the existence of four divisions in the language: IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE. A source program contains information in all four divisions. The IDENTIFICATION Division simply provides the identification of the program and programmer. The ENVIRONMENT Division allows the user to indicate a great many features which relate to the hardware, and permits the connection between the source program and the hardware.

Almost by definition the ENVIRONMENT Division for a particular program would have to be rewritten if the program was moved to another machine. The intent of the ENVIRONMENT Division is to allow all hardware-dependent elements to be encapsulated in one place.

The DATA Division contains facilities to describe logical files and logical records and allows some user control over the object-time representation of the data. This is a greatly expanded and powerful version of the concept of "data declarations" in other languages. This division is relatively machine-independent, but not entirely so if efficiency of the internal data storage is a major consideration (which is normally the case). Thus it may be necessary to rewrite this division in going from one machine to another, although that can be avoided at the expense of some object-time efficiency. A single DATA Division can be written and used for any source program which uses the same files.

The PROCEDURE Division allows the user to specify the operations which are to be carried out in a particular program. This corresponds to most of what is normally thought of as "the program" in other languages. The PROCEDURE Division is machine-independent providing the user does not deliberately go out of his way to do things which are machine-dependent, and avoids the few elements in this division which are hardware-dependent (e.g., quirks of arithmetic calculation).

It is important to note that these four divisions are the basic structure of the language; that is a very different concept than the way in which the ANSI standard has been constructed. The COBOL 74 Standard is composed of a nucleus and 11 data processing modules. This concept is not particularly important with regard to evaluating COBOL against TINMAN requirements, except that it makes it easy to pinpoint the two modules which are clearly not needed by TINMAN. They are: Sort-Merge and Report Writer. Parts of the other nine modules are also not needed to satisfy TINMAN requirements.

Emphasis on Data and File Handling

The basic purpose of COBOL is to allow efficient handling and minor computation of masses of data which are assumed to be collected on files or mass storage devices. This is what has been classically known as Business Data Processing. Because of the great desire to maintain machine independence, a very important concept in dealing with data is what is known as the

"standard data format". This simply means that from the language point of view and from the user point of view, data will be considered to be in decimal form. This does not require the implementer to store it that way, but allows the user to deal with the data in a machine-independent way.

Naturally, there is a significant orientation towards having effective input/output facilities to deal with the masses of data.

### General Approach to Syntax

From its inception, the emphasis in COBOL has been on readability and the desire to make the language as "English-like" as possible. This has been considered an extremely important characteristic of COBOL, as contrasted with a different concept of including and developing an elegant-language structure and elegant-language features. (See related comments in last paragraph of the section below.)

## IMPORTANT CHARACTERISTICS RE USAGE AND DEVELOPMENT

### Usage and General Views

COBOL is probably the most widely used language, although it is difficult to obtain meaningful statistics on any type and measure of usage. However, the existence of many smaller machines which are used for business data processing tends to support the data (which is not very reliable) but which seems to lead to the conclusion that COBOL is the most widely used language today. It has been implemented on most machines, including very small ones. As is well known, some of the computers that are being called mini-computers in 1976 are more powerful than some of those existing in the early 1960's and COBOL was implemented on many of those early machines. This is important simply because it proves that COBOL can be implemented effectively on almost any machine.

Unfortunately, COBOL is not well thought of by computer scientists or by most language experts. They tend to be disdainful of the English-like nature of the language and its resulting wordiness just as they dislike the inelegant program structure. As in the case of FORTRAN, COBOL's basic mode was set very early (namely in 1959), and while numerous improvements have been made to the language since then, the

framework established at that time has remained fundamentally unchanged.

### Development

From COBOL inception in 1959, the language specifications have been under the jurisdiction of a fairly loosely defined organization called CODASYL which has been the standard maintenance body. In addition, under the auspices of the American National Standards Institute, COBOL was first standardized in 1968 and then later a revised standard was developed in 1974. The evaluation with respect to TINMAN in this report is based on the 1974 Standard, with just a few comments on the 1968 version. A newer version (1976) of the language was defined but is not standardized by ANSI.

## GENERAL EVALUATION OF COBOL

### Strengths

The basic strengths of COBOL considered as a language are the following:

1. Flexible data formats

2. Detailed control of the data formats is available to the programmer, but these details do not have to be specified; some defaults are available

3. Very powerful and flexible data description which could easily be extended to handle new data types

4. Good input/output, particularly allowing for both sequential and random access and mass storage files

5. Readability - COBOL is verbose, but this makes it easily readable to anyone knowledgeable in data processing

6. Probably the most widely used language in existence

today (although this is impossible to either define or measure specifically)

7. Has been widely implemented on most machines, which means that it could probably be implemented on new central processors which might be developed.

Weaknesses

The basic weaknesses of COBOL, considered as a language, are the following:

1. Program structure is very weak. Essentially (only) two levels of procedure hierarchy are allowed, but there is no provision for separate data declarations at different levels.

2. Parameter passage for procedures is very weak.

3. No functions are allowed and this might be hard to add syntactically. (Note that the intended purpose of COBOL is for an area of application where functions are not common nor very necessary.)

4. No extensibility features are in the standard (although they could be added).

5. No pointer mechanisms are available.

SPECIFIC EVALUATION WITH REGARD TO TINMAN

The evaluation of TINMAN with respect to COBOL 68 was done after the evaluation with respect to COBOL 74. Since most implementations of COBOL are of the COBOL 68 Standard, there is some importance associated with an evaluation based on COBOL 68. On the other hand, the long time-span between the first (i.e., 1968) standard, and the more current COBOL 74 version makes the COBOL 74 evaluation far more important in terms of what COBOL actually is today. It is worth pointing out that there is a 'CODASYL COBOL Journal of Development' (1976) which contains still more changes and additions to the COBOL 74 Standard. However, no evaluation of TINMAN was made with respect to the 1976 JOD, and it is believed that it would not significantly change the overall evaluation of COBOL.

In order not to clutter up the main part of the evaluation which is with respect to COBOL 74, the only time that comments are made with respect to COBOL 68 is when the evaluation due to COBOL 68 is lower from that of COBOL 74. If the evaluation remains the same (although the reasoning may differ in the two cases), no comments are made about COBOL 68.

The following are paragraph-by-paragraph comparisons of the TINMAN requirements to the candidate language specifications document:

A1.COBOL.Typed Language
T (Satisfies)

The Data Description (page I-119) in the COBOL DATA DIVISION ensures that this requirement is fully satisfied. For each item of data and for all groupings of data, their types must be fully described in the DATA DIVISION. This is done largely (but not entirely) through the PICTURE clause (pages II-18 through II-26).

A2.COBOL.Data Types
Partially Satisfies

The fundamental concept of describing data in COBOL is different from most other languages. COBOL bases its philosophy (for numeric data types) on the concept of a "standard data format" which is essentially a conceptual decimal representation of numbers. Therefore, integers and fixed-point numbers are easily described simply by indicating the position of the decimal point which is done in the COBOL PICTURE clause (pages II-18 through II-26). There is no floating-point in COBOL. The COBOL standard does not allow Boolean data types, although the "CODASYL Journal of Development" does. However, COBOL allows many types of "conditions" which serve similar purposes, i.e., have values of True or False (see pages II-41 through II-49). Character data type variables are allowed and in COBOL are referred to as alphanumeric (page II-22). Arrays are allowed (although limited to 3 dimensions) by using the OCCURS clause (pages III-2 through III-4). Records are definitely allowed, and in fact form the heart of the concept of organizing data for COBOL programs. They are shown through the use of the level number concept (page II-17).

Floating point exists in at least one compiler and possibly in others; i.e., it is not difficult to do, either from a language or an implementation viewpoint. Since the "CODASYL Journal of Development" includes Boolean data types, the language specifications already exist although this was not included in the standard. From a language point of view, it is trivial to lift the restriction of arrays to 3 dimensions; the implementation techniques for multidimensional arrays are well known although they may not be used in current COBOL compilers.

A3.COBOL.Precision
Partially Satisfies

COBOL does not allow floating-point arithmetic and therefore does not deal with the problem of the precision of floating-point arithmetic calculations. The requirement for precision specification for individual variables is partially met. The precision of each numeric data item is automatically defined as part of the data description of that variable, and in particular by the PICTURE clause (pages II-18 through II-20). In doing arithmetic calculations, a maximum of 18 digits is allowed for computational purposes. It is therefore formally true that the programmer does not have complete control over the precision desired, but on a more practical basis 18 decimal digits would seem to suffice in most cases. The rules associated with the COBOL arithmetic verbs (ADD, SUBTRACT, MULTIPLY, DIVIDE and COMPUTE) all require that the compiler provide enough precision to satisfy the constraints on the precision desired in the result. (See for example ADD, page II-56, point (5).)

For needed changes, note the following: if floating point were added then precision rules could also be included. The limitation on 18 digits could be increased in the language specifications if desired and this would merely require compilers to provide greater precision in their calculations.

A4.COBOL.Fixed-Point Numbers
T (Satisfies)

It is not entirely clear whether COBOL fully or partially satisfies this requirement, since the requirement itself is ambiguous. In general terminology, fixed-point numbers are sometimes interpreted as simply numbers between plus and minus 1, whereas in other cases fixed-point numbers are numbers of the form NNN.NNN with any number of digits on either side of the decimal point. Because of the COBOL philosophy of treating all numbers as decimal, the user specifies the location of the decimal point in the PICTURE clause (pages II-18 through II-20) that will automatically determine the fractional step size. In this context, scale factor management is not really relevant. However, it should be noted that all arithmetic computations are to be implemented essentially by alignment on the decimal point (see page II-51, section 5.3.4). Rounding is controlled by the programmer (pages II-50 and II-55). Integers will, indeed, be treated as exact quantities for the reasons just given.

A5.COBOL.Character Data
T (Satisfies)

The COBOL character set consists of 51 characters, all included within the ASCII Code. The user is allowed to declare a collating sequence as part of the description of the object computer (page II-6). If the collating sequence is specified, it will be used to determine the

truth value of any non-numeric comparisons; if it is not specified, then the collating sequence inherent in the Object computer will be used. The CODE-SET clause (page IV-12) specifies the character code used externally and the algorithm for converting from external to internal usage.

COBOL 68 - Partially satisfies. COBOL 68 does not allow the user to specify the collating sequence.

A6.COBOL.Arrays
T (Satisfies)

COBOL fully satisfies the requirement as written, although COBOL has a number of restrictions which are not prohibited by the requirement. COBOL allows specification of the number of dimensions in the OCCURS clause in the Data Description (pages III-2 through III-4). The range of subscript values are specified in that clause. The type of each array component is specified in the Data Description of the array. The actual maximum subscript value can either be specified at compile time, or can be determined at object time if the programmer uses the DEPENDING ON clause in the OCCURS clause. The use of the OCCURS clause requires the programmer to specify the number of dimensions at compile time. The lowest subscript value must be the number 1, i.e., it cannot be zero or negative (page III-2, section 2.1.3, point (2)). It should be noted that the subscript must be an integer, or a variable which is declared as an integer, i.e., the subscript itself cannot be an expression. Furthermore, subscripts cannot be subscripted (page I-89, section 5.3.3.8.2).

Only three subscripts are allowed (page I-89); however, it would be a trivial language change to permit any number. This requirement is a holdover from the earliest days, and also a reflection of the fact that more than 3 dimensions is not that necessary in normal business data processing problems. The ranges of subscript values are specified in the OCCURS clause. It is worth noting that COBOL also provides a method for handling arrays known as "indexing". This simply means that variables defined as indexes are permitted, but they are not treated as other variables, and are handled in whatever way is deemed most effective for the particular hardware. The programmer has no control over the method, but does determine when index variables are used.

A7.COBOL.Records
T (Satisfies)

The REDEFINES clause in the Data Division (page II-27) permits records to have alternative structures which are defined at compile time. Specifically, "the REDEFINES clause specifies the redefinition of

a storage area, not of the data items occupying the area" (page II-27, section 4.8.4, point 2). There is a restriction that variable length structures cannot be redefined. Discrimination is done by the proper use of the data names, i.e., the programmer uses the names associated with the structure that he wants. There is also a RENAMES clause (page II-29 through II-30) which permits some regrouping of data, but it is not the main data overlay facility; the REDEFINES is.

B1.COBOL.Assignment and Reference
T (Satisfies)

Assignment in COBOL is carried out from the MOVE statement and the five arithmetic statements (ADD, SUBTRACT, MULTIPLY, DIVIDE, COMPUTE). While this assignment facility does permit any value of a given type to be assigned to a variable, the latter must be "large enough" to receive the full value of the "sending data". Since COBOL variables occupy differing amounts of storage (rather than a word or other size predetermined by the compiler), if the "receiving" data item is "too small" to contain the "sending" item then there are rules in the language specification which determine what is to be done. Reference to any data-name does indeed retrieve the last assigned value.

Since COBOL does not permit encapsulated type definitions, the user is not able to define assignment and access operations within them. The wording of the requirement seems to make it legitimate to consider the requirement fully satisfied because it says essentially "If there are encapsulated type definitions then ...."

B2.COBOL.Equivalence
T (Satisfies)

The "IF condition" statement in COBOL provides for logical identity tests. Numeric operands are compared on the basis of algebraic value, regardless of internal format (page II-42, section 5.2.1.1.1). Non-numeric operands are compared on a character by character basis (pages II-42 through II-43). It is allowable to compare an integer variable or literal to a non-numeric operand (page II-42, section 5.2.1.1.2). This is done by essentially defining the integer as an alphanumeric variable and then comparing two non-numeric operands. This would appear to violate the requirement that elements of disjoint types never be identical, but it seems legitimate to consider this within the "10% deviation" allowed for a "fully satisfies" determination.

For needed changes, note the following: the exception to 100% compliance noted above (i.e., ability to compare numeric and non-numeric operands) could easily be removed from the language.

B3.COBOL.Relationals
T (Satisfies)

The "fully satisfies" evaluation refers to the requirement actually stated in B3 that "Relational operations will be automatically defined for numeric data". It also satisfies the other part of the requirement by default because COBOL does not have types defined by enumeration. As for the requirements stated in the text, COBOL contains the 3 relational operators "greater than", "less than", "equal to" and also has the "not" operator which then permits the same result as having "greater than or equal to", "less than or equal to" and "unequal". As far as inhibiting ordering definitions, there are no types defined by enumeration in COBOL; thus there is no need to be concerned with unordered sets.

B4.COBOL.Arithmetic Operations
T (Satisfies)

See ADD, SUBTRACT, MULTIPLY, DIVIDE and COMPUTE verbs for the basic computation. Division with and without remainder is permitted (page II-61, Format 4). Exponentiation and negation operators exist (page II-39). It should be noted that there are no floating-point variables and hence no arithmetic operations to be applied to them.

For needed changes, note the following: if floating point was to be added as a data type, no change or addition to the syntax of the existing 5 arithmetic verbs is needed.

B5.COBOL.Truncation and Rounding
T (Satisfies)

In the most literal sense of the words, the COBOL specifications violate the truncation rule; however, within any reasonable interpretation of what is intended by the requirement, COBOL satisfies it completely. To be more specific, the whole concept of arithmetic and assign operations on data in COBOL is based on the concept of aligning on the decimal point. Arithmetic is carried out on numbers after alignment on the decimal point. More importantly, when results are assigned (by any of the 5 arithmetic verbs or the MOVE statement) the result is based on the described format of the receiving field based primarily on alignment of the decimal point. Thus, if a programmer chose to store an integer which had 10 digits in it into a field which had only 5 digits in it, then there would indeed be truncation of the 5 most significant digits. However, this is completely under programmer control and the compiler has nothing to do with it whatsoever. It is being assumed that the reasonable interpretation of the requirement is that the compiler cannot perform a truncation of the most significant digits of the numeric quantity of its own volition. Rounding occurs on

the least significant digits and is carried out only under explicit programmer control with the use of the ROUNDED option in the arithmetic verbs.

For needed changes, note the following: there are no floating-point numbers in COBOL, but if they were added these requirements could certainly be satisfied.

B6.COBOL.Boolean Operation
Partially Satisfies

The operations, "and", "or", "not" are supplied (page II-45). There is no "exclusive or" in the standard although it does exist in the "CODASYL Journal of Development" (page III-7 through III-10). It should be noted that although there are no Boolean data types as such, the 3 indicated operators are used between "conditions" which are essentially expressions having a truth value (pages II-41 through II-49). There is no specification in the language about the methodology for evaluating "and" and "or", i.e., the language does not contain a requirement for evaluation in short circuit mode. This is the only reason that COBOL does not fully satisfy this requirement. Note that the requirement asks for a NOR capability but at a 10 November 1976 meeting at IDA, it was agreed that XOR was intended.

For needed changes, note the following: an exclusive or (XOR) can very simply be added, for it already exists in the "CODASYL Journal of Development" (page III-7 through III-10).

A capability to provide short circuit evaluation could also be added very simply.

B7.COBOL.Scalar Operations
T (Satisfies)

Without specifying all of the major details, the COBOL MOVE verb (pages II-74 through II-76) does permit the required transfers. It should be noted that the correspondence will be by position unless the programmer specifies MOVE CORRESPONDING in which case it will be done by alignment of data names.

B8.COBOL.Type Conversion
Fails to Satisfy

The reason that COBOL fails is because the fundamental principle

behind doing calculations on numeric data is alignment on the decimal point independently of the internal representation. Therefore, conversion operations between integers and fixed-point numbers are not specified explicitly by the programmer. Furthermore, the facility to show different "usages" for arithmetic (which refer to internal representations specified by the implementer) means that there are even more implicit conversions! In addition, because of the richness of data types in COBOL (involving alphabetic, alphanumeric, alphanumeric edited, numeric and numeric edited) as described in the PICTURE clause (pages II-18 through II-26) there are numerous rules spelled out as to what conversions are legal and how those are to be carried out automatically in performing computations and/or assignments. These rules are explicitly stated in the language specification and are not at the discretion of the implementer. Furthermore, the programmer is in complete control because any conversions carried out are based on the way he described the data. If "implicit type conversions" means that the data description controls the conversion, then COBOL conversions are all implicit.

For needed changes, note the following: it would not be possible to modify COBOL to meet this requirement because COBOL's current way of handling these conversions is a fundamental part of the COBOL concept.

B9.COBOL.Changes in Numeric Representation
Partially Satisfies

Certainly COBOL does not require explicit conversion operations between numerical ranges. However, the requirement that "There will be a run time exception condition when any integer or fixed-point value is truncated." is ambiguous. It is not clear whether the requirement means that there should be an automatic transfer to some predetermined routine whenever this error occurs, or whether it is meant to say that the programmer can specify what is to happen if this error occurs. The "ON SIZE ERROR" clause in the 5 arithmetic verbs permits the programmer to determine what is to happen if truncation or other errors such as division by zero occur.

For needed changes, note the following: either interpretation would be easy to include in COBOL.

B10.COBOL.Input/Output Operations
T (Satisfies)

Part of COBOL's major strength lies in its powerful I/O capability. The Sequential I/O module permits programmer interaction with sequential files, independent of the particular physical device involved. The connection between the program files and the hardware is made by the

user through what he writes in his program in the ENVIRONMENT DIVISION and specifically in the File Control paragraph of the Input/Output section (pages IV-4 through IV-5). The verbs READ and WRITE permit input and output of logical records respectively. The ACCEPT and DISPLAY verbs are used for low-volume data input and output respectively and, in particular, can be used effectively for terminals. In addition to this, there is a Relative I/O module which provides the capability of accessing records of a mass storage file in either a random or sequential manner.

There are a number of ways in which the user has control over exception conditions. In particular, the AT END condition used in the READ (page IV-28) and WRITE (page IV-34) verbs allows the user to specify what is to be done when the end of file is reached. There is a USE statement (pages IV-32 through IV-33) which permits the user to specify procedures for the input/output error handling that are in addition to the standard procedures provided by the input/output control system. Specifically, the user can specify the program steps to be taken upon occurrence of an error or an exception. This is not installation-dependent; however, the user is able to specify in the ENVIRONMENT DIVISION the relation between the logical files and the physical hardware that is involved. Furthermore, the I/O-CONTROL paragraph (pages IV-6 through IV-8) permits the user to specify how reruns are to be handled.

The only aspect of this which COBOL seems to fail to meet 100% is the ability to dynamically assign and reassign devices; i.e., there is no way the programmer can change the assignment of physical devices at execution time.

For needed changes, note the following: the language could be modified to permit dynamic allocation of devices, but it would probably require major changes in the concept of the ENVIRONMENT DIVISION and would certainly increase implementation difficulties.

B11.COBOL.Power Set Operations
Fails to Satisfy

COBOL does not allow power sets of data types, nor enumeration types.

For needed changes, note the following: it would be moderately difficult to add these facilities.

C1.COBOL.Side Effects
T (Satisfies)

The COBOL standard states with respect to arithmetic expressions that "when the sequence of execution is not specified by parentheses, the order of execution of consecutive operations of the same hierarchical level is from left to right" (page II-40, point 2).

C2.COBOL.Operand Structure
T (Satisfies)

The operator hierarchies involved pertain to (1) the arithmetic operations, (2) the logical operations and (3) the relationship between them. There are the obvious 4 levels for arithmetic operation, the obvious 2 levels for logical operation (pages II-39 and II-49, respectively). However, there are several types of conditions (which are really Boolean expressions although called "conditions" in COBOL) which also have a hierarchy (page II-48). Parentheses are of course allowed in all the appropriate places, either to overcome the normal hierarchy, or to specify an execution order within elements of the same precedence level. The condition evaluation rules, when both arithmetic and conditional expressions are involved, are well defined (pages II-47 through II-49). The user is not able to define new operator precedence rules, nor to change the precedence of existing operators.

C3.COBOL.Expressions Permitted
Fails to Satisfy

COBOL has a great many restrictions on the use of expressions. For example, subscripts cannot be expressions, they cannot be subscripted, etc.

For needed changes, note the following: it would be difficult, and perhaps impossible, to modify COBOL to meet this requirement.

C4.COBOL.Constant Expressions
Fails to Satisfy

COBOL allows literals (i.e., constants) in many places but does not permit constant expressions where constants are allowed.

For needed changes, note the following: it probably would be difficult to add this facility because it might make the syntax ambiguous. It would certainly make the parsing more difficult.

C5.COBOL.Consistent Parameter Rules
Partially Satisfies

Since COBOL does not have many instances of parameters, there are not too many rules. There is only one way to invoke a procedure, and this is defined in the Inter-Program Communication module (pages XII-1 through XII-8). The parameters are defined as part of the PROCEDURE DIVISION heading (page XII-4). As far as exception handling is concerned, there is consistency throughout those in the sense that whenever the specific clauses (e.g., ON OVERFLOW, ON SIZE ERROR) are used, each is followed by an imperative statement. Whether this can, or should be, considered a parameter is certainly debatable. Thus, to some extent COBOL satisfies this requirement almost by default.

COBOL 68 - Not applicable. COBOL 68 does not allow subroutines, and hence there are no rules on parameters.

For needed changes, note the following: parameter handling is very weak in COBOL. It would require significant changes to improve it.

C6.COBOL.Type Agreement in Parameters
Fails to Satisfy

While the most accurate judgment would seem to be "fails", it should be pointed out that this is not 100% certain. There appears to be an error in the text of the COBOL standard. The method for handling subroutines in COBOL (Section XII) specifies that the subroutine will contain a Linkage Section which gives the data descriptions for the formal parameters (page XII-2). The specifications say "Within a called program, Linkage Section data items are processed according to their data descriptions given in the called program" (page XII-4). The specification then goes on to refer to the actual parameters, and then says "Their descriptions must define an equal number of character positions; however, they need not be the same name". If the word "name" was not there, the rule would be very explicit that the formal and actual parameters need not be of the same type (which would then violate the TINMAN requirement). However, the wording in the "CODASYL Journal of Development" says "Except that they must define an equal number of character positions, their descriptions need not be the same" (page III-7, point 2). In COBOL terminology, if data items do not have the same description, they are not necessarily of the same type. Thus the COBOL standard seems ambiguous whereas the "CODASYL Journal of Development" is quite clear (and violates the TINMAN requirement). It is not clear from the specifications what happens to data arrays.

COBOL 68 - Not applicable. COBOL 68 does not allow subroutines, and hence there are no rules on parameters.

For needed changes, note the following: it would be fairly easy to change COBOL to meet this requirement.

C7.COBOL.Formal Parameters
Fails to Satisfy

COBOL does not provide a special class of data parameter to serve as a constant. It does provide the second class of data parameter, namely the variable, and the passage of parameters is a CALL by location (page XII-2). There are no formal parameter classes for control action on exception condition nor for procedure parameters. However, as described under B10, COBOL does have very good language facilities for handling exceptions.

COBOL 68 - Not applicable. COBOL 68 does not allow subroutines, and hence there are no rules on parameters.

For needed changes, note the following: it would be quite difficult to modify COBOL to meet this requirement. It would require fundamental changes in concepts.

C8.COBOL.Formal Parameter Specifications
T (Satisfies)

COBOL requires a Linkage Section in the DATA DIVISION of the procedure declaration and this Linkage Section describes the data to be used in the procedure. This must be available at compile time. However, the specifications state "Within a called program, Linkage Section data items are processed according to their data descriptions given in the called program" (page XII-4). Thus the "specification of ... parameters" in the procedure declaration is essentially overridden by the calling program.

COBOL 68 - Not applicable. COBOL 68 does not allow subroutines, and hence there are no rules on parameters.

C9.COBOL.Variable Numbers of Parameters
Fails to Satisfy

The actual names of all the parameters must be shown at compile time.

COBOL 68 - Not applicable. COBOL 68 does not allow subroutines, and hence there are no rules on parameters.

For needed changes, note the following: it would be fairly easy to

add this to the language specifications, but the implementation would have the normal difficulties associated with it.

D1.COBOL.Constant Value Identifiers
T (Satisfies)

There is a facility called the VALUE IS clause (pages II-36 through II-38) which is part of the Data Description and which allows the assignment of specific literals (i.e., constants) to data at compile time. This seems to satisfy the spirit of this requirement. Constants can also be assigned to data items in the Communications Section and to a condition name (I-78 and II-37, section 4.13.5) and to items in the Report Writer Section. Although section and paragraph names cannot have constants assigned to them, that capability doesn't seem to make sense anyway within the overall spirit of TINMAN's avoidance of "tricky" coding.

D2.COBOL.Numeric Literals
T (Satisfies)

COBOL provides for both numeric and non-numeric literals (pages I-80 through I-82). Numeric literals are limited to 18 digits and can contain (only) one decimal point. The internal representation of this literal will be consistent with the internal representation of all other data given in the standard data format of COBOL. COBOL also allows for non-numeric literals of up to 120 characters in length. There is currently no rule in COBOL that says "numeric constants will have the same value in both programs and data" but it would be a trivial rule to impose.

The requirement that "numeric constants will have the same value .. in both programs and data..." is a directive to the implementer and not a language specification. There is no such statement in the COBOL specifications but it would be trivial to include it.

D3.COBOL.Initial Values of Variables
T (Satisfies)

The VALUE IS clause (pages II-36 through II-38) enables the user to specify the initial values of variables as part of their declaration, but does not require him to do so. There are no default initial values in COBOL. Because of the fundamental way in which COBOL approaches data it is meaningless to have a special provision "for run time testing for initialization".

D4.COBOL.Numeric Range and Step Size
T (Satisfies)

The PICTURE CLAUSE (pages II-18 through II-19) for numeric items allows (and requires) the user to specify the number of decimal digits involved.

The number of occurrences of the digit "9" combined with the presence of an actual or assumed decimal point indicate the maximum number of digits and hence the range of the numeric variables. As an example, a "picture" of 99999V9999 would represent a number that has four decimal places and a whole value of less than 100,000; the V represents the location of an assumed decimal point. If a period is used, it is treated as an actual decimal point; thus replacing V by "." represents numbers of the same range. The step size is automatically the lowest digit position, i.e., the digit position furthest to the right of the decimal point. (In the above case, the step size is .0001.) If the assumed decimal point is actually outside the number of significant digits (represented by the string of 9's) the user specifies this by using a string of P's on the right or left as appropriate. Thus 99 PPP represents numbers from 00000 to less than 100000 with only 2 significant digits. Numbers cannot be more than 18 decimal digits long.

D5.COBOL.Variables Types
T (Satisfies)

Arrays can be components of records via the OCCURS clause (page III-2) and records can be components of arrays (page III-2). Data items can be given any range of values.

D6.COBOL.Pointer Variables
Fails to Satisfy

COBOL does not have any pointer variables.

For needed changes, note the following: it would be difficult, if not impossible, to add a pointer mechanism to COBOL.

E1.COBOL.User Definitions Possible
Partially Satisfies

This evaluation is valid if and only if subroutines are considered a form of defining new operations. If not, then COBOL fails completely on this requirement.

For needed changes, note the following: the ability to add new syntactic operations (i.e., new verbs and not just subroutines) is conceptually very easy; in fact, from its earliest published specification (1960) until 1968, COBOL had a DEFINE verb which enabled the user to create a new verb with specified syntax which could then be used just as any other verb. DEFINE was eventually dropped from the language specification primarily because nobody had implemented it. There are some commercial software houses which have developed add-on packages to COBOL which permit the equivalent of this and so it can be implemented.

Defining new data types in terms of the base language would probably not be very difficult because of the existing power and complexity of the existing COBOL data types.

E2.COBOL.Consistent Use of Types
Fails to Satisfy

COBOL does not enable the user to define new data types.

For needed changes, note the following: defining new data types in terms of the base language probably would not be very difficult because of the existing power and complexity of the existing COBOL data types.

E3.COBOL.No Default Declarations
Fails to Satisfy

COBOL does have default declarations. At least the following have been noted: if the USAGE clause is not specified, the usage is implicitly DISPLAY (page II-35, point (5)). In the SYNCHRONIZE clause, if neither LEFT nor RIGHT is specified, then it is assumed that the item will be positioned in an efficient manner as determined by the implementer (page II-33, point (2)); in the SIGN clause, if the SEPARATE CHARACTER phrase is not present, then certain other assumptions are made (pages II-31 through II-32, point (3)). There may be other defaults as well.

For needed changes, note the following: the elimination of all these defaults is very simple; one merely specifies exactly what is to happen in each existing default case and requires the appropriate item to be put into the language specification. An alternative way of handling it is to remove the optional status of these clauses and include another alternative to those already existing.

E4.COBOL.Can Extend Existing Operators

Fails to Satisfy

COBOL fails by default to satisfy this requirement because it does not allow the creation of new data types.

For needed changes, note the following: defining new data types in terms of the base language probably would not be very difficult because of the power and complexity of the existing COBOL data types.

E5.COBOL.Type Definitions
Fails to Satisfy

COBOL fails by default because it does not allow type definitions.

For needed changes, note the following: this probably would not be very difficult to include in COBOL because of the existing power and complexity of COBOL data types. However, even if this requirement could be met, it would then cause difficulty in complying with the other Definition Facility requirement (i.e., category E).

E6.COBOL.Data Defining Mechanisms
Fails to Satisfy

COBOL fails by default because it does not permit the user to define new types.

For needed changes, note the following: this probably would not be very difficult to include in COBOL because of the existing power and complexity of COBOL data types. However, even if this requirement could be met, it would then cause difficulty in complying with the other Definition Facility requirement (i.e., category E).

E7.COBOL.No Free Union or Subset Types
T (Satisfies)

COBOL fully satisfies this by default, i.e., it does not permit type definitions at all and hence does not permit them in a way that is deemed undesirable.

E8.COBOL.Type Initialization
Fails to Satisfy

COBOL fails by default because it does not permit the user to define a new type.

For needed changes, note the following: this probably would not be very difficult to include in COBOL because of the existing power and complexity of COBOL data types. However, even if this requirement could be met, it would then cause difficulty in complying with the other Definition Facility requirement (i.e., category E).

F1.COBOL.Separate Allocation and Access Allowed
Fails to Satisfy

This requirement is really meaningful only in a language which has "own variables" and a true block structure. Since COBOL does not have that type of program structure, this requirement would not be needed in COBOL and hence is not there. Some control over storage and overlay are provided by the Segmentation facility.

For needed changes, note the following: it would be neither possible nor sensible to try to put this into COBOL.

F2.COBOL.Limiting Access Scope
Partially Satisfies

COBOL does not allow type definitions so problems of access in that connection do not arise. It is possible to associate new local names with separately defined programs. This is done via the Linkage Section of the Inter-Program Communication Module (page XII-4).

For needed changes, note the following: it would be very difficult to add the missing features to COBOL.

F3.COBOL.Compile Time Scope Determination
T (Satisfies)

The term "scope" is not used in COBOL and is not really applicable to COBOL as it is to the block-structured languages such as ALGOL and PL/I. However, COBOL definitely satisfies the requirement that "the scope of identifiers will be wholly determined at compile time". This is achieved through the rules on qualification for both data names and procedure names (pages I-87 through I-89).

F4.COBOL.Libraries Available

T (Satisfies)

COBOL permits the handling and the use of libraries through the Inter-Program Communication module, which allows procedure definitions for routines which are to be separately compiled, and allows a CALL statement in the basic program. Both procedures and data descriptions can be obtained from a library and included directly in a source program via the COPY statement in the Library Module (pages X-2 through X-4). The development and availability of libraries is of course beyond the scope of a language evaluation; however, nothing in COBOL prevents the development of such libraries.

COBOL 68 - Partially satisfies. Only one library can be accessed from within a single program, and there is no provision for subroutines with parameters.

F5.COBOL.Library Contents
T (Satisfies)

COBOL allows the existence of Libraries through both the Library and the Inter-Program Communication module. In addition, the ENTER verb (page II-63) provides a means of allowing the use of languages other than COBOL.

F6.COBOL.Libraries and Compools Indistinguishable
T (Satisfies)

COBOL contains a COPY statement (page X-2). This COPY statement can be used for both programs and data descriptions, and can be used almost anywhere in the source COBOL program. The specifications state "COBOL libraries contain library texts that are available to the compiler for copying at compile time. The effect of the interpretation of the COPY statement is to insert text into the source program, where it will be treated by the compiler as part of the source program" (page X-1). This applies to both procedures and data descriptions as well as other elements of a COBOL program.

F7.COBOL.Standard Library Definitions
T (Satisfies)

In the CONFIGURATION Section of the ENVIRONMENT DIVISION there is a standard format for describing the source computer, the object computer, various hardware switches, etc. (page I-113). COBOL does contain a standard way of accessing particular pieces of hardware such as a real-time clock through the ACCEPT verb. In the FILE CONTROL paragraph,

the user is allowed to specify a file RERUN EVERY integer CLOCK UNITS. Furthermore, the Communication module provides standard ways of dealing with message control systems (page XIII-3).

COBOL 68 - Partially satisfies. There is no provision in the ACCEPT verb of COBOL 68 for obtaining information from the real-time clock. Furthermore, COBOL 68 does not contain the Communication module and hence no way of dealing with message control systems.

G1.COBOL.Kinds of Control Structures
Partially Satisfies

COBOL has normal sequential control through the executable statements. The conditional facility is handled by the IF statement (pages II-66 through II-67), as well as by transfers of control which occur when exceptions are handled. Iterative control is provided by the PERFORM statement (pages II-78 through II-84). COBOL does not allow recursive routines. This is implied as part of the CALL statement specifications which say "On all other entries into the called program, the state of the program remains unchanged from its state when last exited. This includes all data fields, the status and positioning of all files, and all alterable switch setting." (page XII-5, point 3). (The "all other entries" refers to the initial call or after a CANCEL verb is executed.) COBOL does not provide control structures for parallel processing, except that the input/output operations may be implemented for parallel execution without any further action by the user. COBOL does provide a great deal of exception handling. In particular, COBOL allows the programmer to specify what will happen in the case of a size error in arithmetic computation, an overflow of memory in calling a subroutine, end of file, and several more specialized cases. (Partial list is on page I-102.) In addition, the USE statement (pages IV-32 through IV-33) allows the user to specify procedures for input/output error handling in addition to the standard ones provided by the system.

For needed changes, note the following: to permit recursion in COBOL is no more difficult than in any other language as long as one is willing to pay the penalty at object time. It would be very easy to have the language require that a subroutine be declared recursive as part of its heading, and this would permit the implementer to avoid including the object time package to deal with non-existent recursion. The parallel processing could probably be added relatively easily.

G2.COBOL.The GOTO
Partially Satisfies

COBOL contains the simple GOTO statement (page II-65). Since there

are no scope rules in the sense of the block-structured languages, the GOTO can be used to transfer control to labels anywhere in the program. However, there is an ALTER statement (page II-57) which allows the change of the label in the "GOTO label" at object time. This unfortunately "encourages its use in dangerous and confused ways".

For needed changes, note the following: it would be trivial to bring COBOL into full compliance with this requirement by eliminating the ALTER statement. In fact, the "CODASYL Journal of Development" for 1976 has removed ALTER from the language.

G3.COBOL.Conditional Control
Partially Satisfies

The basic conditional control statement in COBOL is the IF (pages II-66 through II-67). This partially violates the requirement that the language specify that an ELSE clause appear after each IF...THEN, because under certain conditions the ELSE can be eliminated - namely when control is to go to the next sentence. The "condition" following the IF is extremely general (pages II-41 through II-46). The user is allowed to make fairly standard relational tests of size among arithmetic expressions, but can also (1) test the class conditions (i.e., whether the operand is numeric or alphabetic), (2) test a condition name (which specifies specific values which can be tested against), (3) test switch status (refers to the on or off status of an implementer-defined switch and (4) use a separate sign test to determine whether an arithmetic expression is positive, negative or zero.

For needed changes, note the following: it would obviously be a trivial language change to require the existence of an "ELSE" after each "IF...THEN".

G4.COBOL.Iterative Control
T (Satisfies)

The PERFORM verb is the basic iterative control statement. It does permit the termination condition to appear anywhere in the loop. The PERFORM statement allows iteration on three nested variables with a different termination condition for each one all in one PERFORM statement. Entry is allowed only at the head of the loop. The second format of the PERFORM statement allows the user to specify a fixed number of iterations. The language specifications do clearly indicate the value of a control variable after termination (see pages II-82 through II-83).

G5.COBOL.Recursive Routines

Fails to Satisfy

COBOL does not allow recursive routines. This is stated as part of the CALL statement specifications which say (page XII-5, point 3) "On all other entries into the called program, the state of the program remains unchanged from its state when last exited. This includes all data fields, the status and positioning of all files, and all alterable switch settings." (The "all other entries" refers to the initial call or after a CANCEL verb is executed.)

For needed changes, note the following: to permit recursion in COBOL is no more difficult than in any other language as long as one is willing to pay the penalty at object time. It would be very easy to have the language require that a subroutine be declared recursive as part of its heading, and this would permit the implementer to avoid including the object time package to deal with non-existent recursion.

G6.COBOL.Parallel Processing
Fails to Satisfy

COBOL does not contain any user-controlled parallel processing capability.

For needed changes, note the following: it is probably not too difficult to add parallel processing to COBOL.

G7.COBOL.Exception Handling
T (Satisfies)

COBOL has very good exception handling. A list of exception cases that the user can control is contained in the "Conditional Statement" list on page I-103. In addition, there is a USE verb which permits special handling of files beyond the normal system defined facilities. In each of these "exception cases", the user specifies the statement that is to be executed in the exception situation. This statement can be either a GOTO, or a call to a library routine, or a single statement.

G8.COBOL.Synchronization and Real-Time
Partially Satisfies

COBOL only partially satisfies this requirement because it does not have the fundamental capability of handling parallel processing. The ACCEPT verb allows the user access to Date, Day and Time from the hardware. There is a Communication module which provides the user with

considerable facility for the specialized application of teleprocessing (see pages XIII-1 through XIII-23). In this facility it is assumed that there is a Message Control System (MCS) beyond the reach of the programmer. The purpose of this particular facility is to provide the interface between the source program and the user. The system allows the user to receive and send messages to enable and disable the appropriate input/output devices.

For needed changes, note the following: considering the overall structure of COBOL and the ability of the user to make certain specifications in the ENVIRONMENT DIVISION, it would probably be quite possible and feasible to add this facility without disturbing the overall structure of COBOL from a syntactic point of view.

H1.COBOL.General Characteristics
Partially Satisfies

COBOL is certainly the most readable of all programming languages. It satisfies many, but not quite all, of the specific requirements of H1. Dealing with each one in turn, the following comments should be made. The source language is partially free format. It is only partial because there is a Reference Format in which certain subunits of the program must start in certain columns. Aside from that, the programmer may write in any form that he chooses. (The full details of the Reference Format are found on pages I-105 through I-108.) COBOL does not have a specific statement delimiter, but it does have a sentence delimiter. A sentence is composed of sequential statements. Thus, one determines the beginning of a new statement by the fact that it starts with the appropriate key word, normally a verb. The distinction between a statement and a sentence is important in certain flow-of-control situations. However the user can write each statement as a separate sentence if he chooses in which case this part of the requirement is met.

COBOL certainly allows mnemonically significant identifiers since they are allowed to be up to 30 characters with imbedded hyphens allowed for easy readability.

There are a few abbreviations of key words allowed, but they are specified in the syntax itself. For example, CORRESPONDING can be abbreviated as CORR, but unless the syntax specifically shows a shortened version of a long word (which makes the short version a reserved word) no abbreviations are allowed. COBOL is syntactically unambiguous.

For needed changes, note the following: it would be difficult - and perhaps impossible - to make COBOL completely free format.

H2.COBOL.No Syntax Extensions
T (Satisfies)

COBOL does not permit the user to create any syntax extensions.

H3.COBOL.Source Character Set
T (Satisfies)

COBOL uses a 51-character set (see page I-54). All of these characters are contained within the 64 character ASCII subset.

H4.COBOL.Identifiers and Literals
Partially Satisfies

COBOL allows identifiers of up to 30 characters. They are composed of letters, digits, and the hyphen. The hyphen can be used anywhere except at the beginning and the end; thus the hyphen serves as a break character (see pages I-76 through I-77). Literals are permitted and are of two types - numeric and non-numeric (pages I-80 through I-81). The numeric literals are limited to 18 digits in length which is consistent with the rules involving arithmetic in COBOL. Non-numeric literals are limited to 120 characters in length. There is no break character used internally for literals; thus, a hyphen or space would be considered as its own specific character within a literal. There is no requirement for separate quotation around each line of a long literal; thus literals can be continued on a second line.

For needed changes, note the following: it would be trivial to impose the rule of requiring literals not to exceed one line, i.e., require separate quoting of each line. However, it would be extremely difficult to impose a rule requiring or permitting a break character within a literal because there is no way then to use that break character as a specific character itself. The common break characters of space and hyphen already have very specific syntactic roles in COBOL and changes in those rules would change much of the syntactic structure.

H5.COBOL.Lexical Units and Lines
Fails to Satisfy

COBOL does not contain a provision prohibiting the continuation of lexical units cross lines. It specifically allows this split of lexical units between lines (see the Reference Format discussion on page I-106, section 5.8.2.2.).

For needed changes, note the following: it would be trivial to include such a rule in COBOL.

H6.COBOL.Key Words
Partially Satisfies

The COBOL key words are reserved; i.e., they cannot be used by the user nor by the implementer (see page I-79, section 5.3.2.2.1.3). Key words are not usable in contexts where an identifier can be used. While COBOL has a very large number of reserved words by actual count - specifically about 300 (see page I-109 through I-110); this is really not nearly as bad as it might seem on the surface. Each of these words appears in very specific places in a very logical and readable way in the formats, but there is not likely to be any difficulty to the user in avoiding these unless he wants to go out of his way to make life difficult. The key words are extremely informative and quite specific in their intent and content. It should be pointed out that in COBOL there are actually three significant categories of words - reserved words which cannot be used by anybody; system names which are defined by the implementer; and user-defined words which cannot be either of the other two classes. There is no problem on selection or portability with the reserved words, since they are clearly listed; on the other hand, since the system name words are defined by the implementer, this could conceivably cause difficulty in portability and compatibility (see pages I-76 and I-78).

For needed changes, note the following: it would be absolutely impossible to reduce the reserved word list to a small number without simultaneously destroying the structure and psychology of COBOL.

H7.COBOL.Comment Conventions
Partially Satisfies

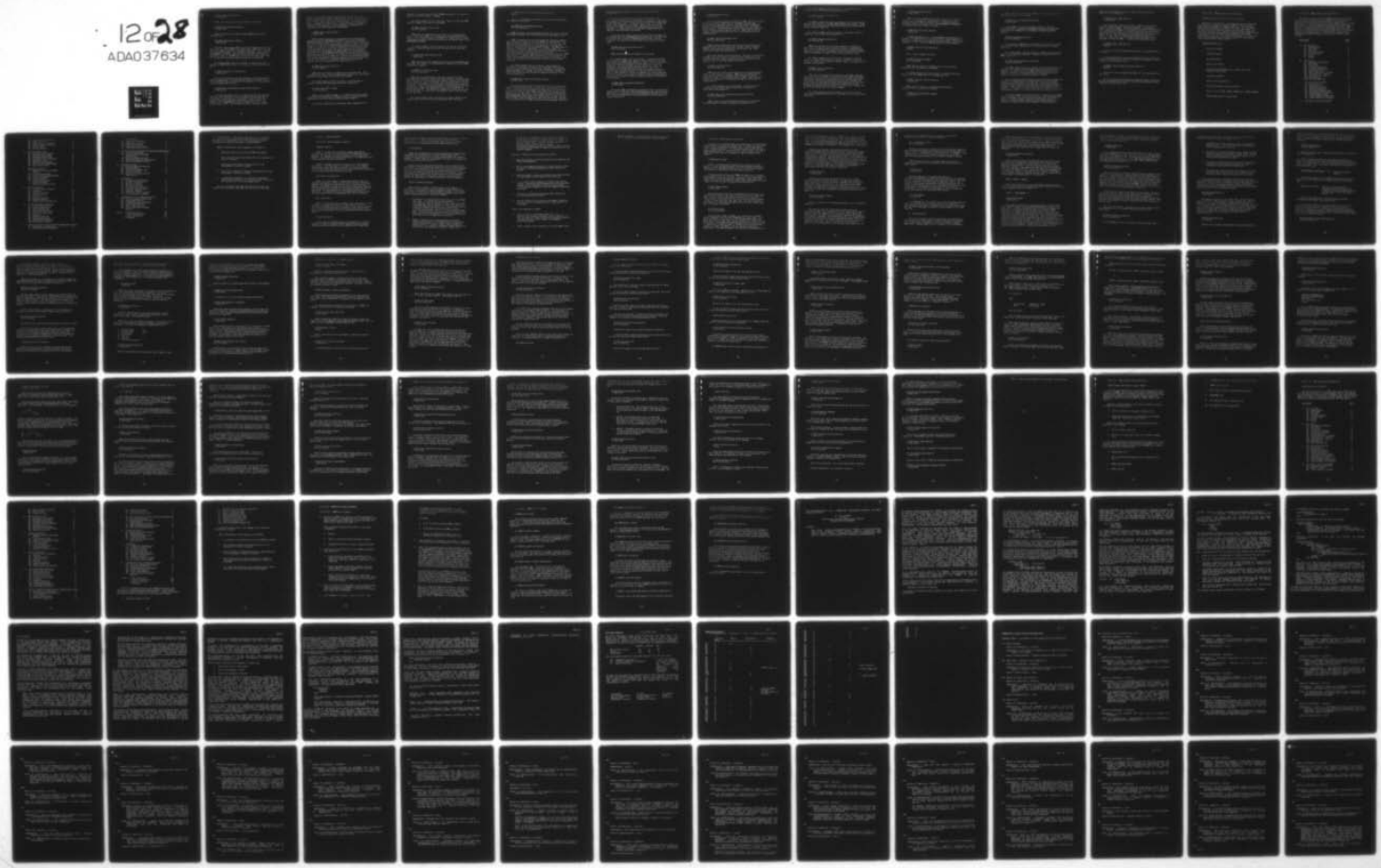
Comments in COBOL are defined by having an asterisk in the continuation indicator portion of a line (page I-108). This prevents the comment from being interspersed in the middle of other portions of the program; i.e., each comment must start on a new line. This of course provides automatic termination of the end of the line but, by putting an asterisk in the next line, comments can run over as many lines as desirable. This does not prohibit automatic reformatting of programs.

For needed changes, note the following: it probably would be difficult to allow comments interspersed within the program other than as a separate line.

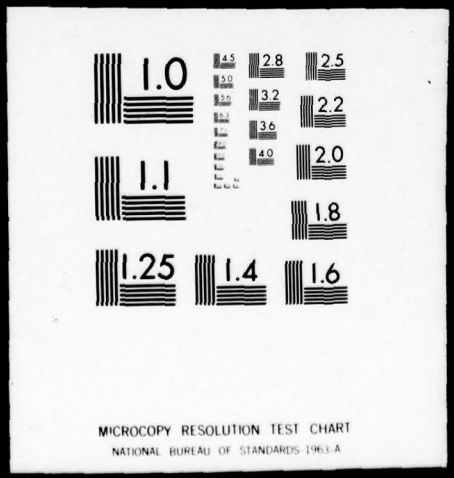
AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

12 OF 28
ADA037634



12 OF 28
ADA037634




tions Optional
13. Compile Time Variables

H8.COBOL.Unmatched Parentheses
T (Satisfies)

COBOL does not permit unmatched parentheses in any form.

H9.COBOL.Uniform Referent Notation
T (Satisfies)

COBOL satisfies this by default because COBOL does not allow function calls.

H10.COBOL.Consistency of Meaning
T (Satisfies)

The only place in which COBOL fails this requirement at all is the use of the "=" in the COMPUTE verb (page II-58). COBOL allows the user to write "COMPUTE *identifier = arithmetic expression*". This is a form of assignment statement. Since assignment statements rarely occur in the COMPUTE verb, but are derived from the use of the four individual arithmetic verbs, or the MOVE verb, this is not very significant. The "=" is used for a relational operator except for this one case.

For needed changes, note the following: it would be trivial to change the COMPUTE verb format, and replace the "=" by a key word, e.g., AS.

I1.COBOL.No Defaults in Program Logic
T (Satisfies)

There are no defaults which the implementer can specify and which affect program logic. There are implicit transfers of control (e.g., from the PERFORM verb or from a CALL) but these are all clearly defined in the language specifications.

I2.COBOL.Object Representation Specifications Optional
T (Satisfies)

It is actually quite difficult to determine whether any language satisfies this requirement unless the language designers describe all the defaults in one place and in some detail. However COBOL does provide many object representation defaults which are useful. For example, in the SYNCHRONIZED clause of the Data Description, (pages II-33 through II-34) if neither RIGHT nor LEFT is specifed, then the

implementer determines the best positioning of the data item. As another example, in the READ verb (page IV-28) the user has a choice of specifying the name of the area into which the records are to be put or merely leaving them essentially in an input buffer. Furthermore, the matter of whether the "next record" is physically moved or whether an internal pointer mechanism is used is entirely in the hands of the implementer, as are all details of I/O handling.

I3.COBOL.Compile Time Variables
T (Satisfies)

COBOL probably satisfies this requirement better than any other language through its ENVIRONMENT DIVISION. Specifically, the OBJECT COMPUTER paragraph (pages II-6 through II-7) allows the user to specify not only the name of the computer but its memory size, and its collating sequence. In addition, the SPECIAL NAMES paragraph (pages II-8 through II-10) allows the user to specify the character set to be used as well as various types of switches which can be tested during the course of the execution of the program. The INPUT-OUTPUT Section (pages IV-4 through IV-5) in the ENVIRONMENT DIVISION allows the user to specify a great deal about the files and their relationship to the actual hardware and internal buffering at object time. Compile time variables are dealt with through the concept of condition names and the ability to test the "on" or "off" status of an implementer-defined switch (page I-78, section 5.3.2.2.1.1.1 and page II-44, section 5.2.1.4).

I4.COBOL.*Conditional Compilation*
Fails to Satisfy

COBOL does not provide this capability in any general way. There is a particular case - namely the Debug Module (Section XI) - which does provide a form of compile time conditional statement.

For needed changes, note the following: it would be easy to include this once the requirement was more fully specified.

I5.COBOL.Simple Base Language
Fails to Satisfy

COBOL is not a simple language. It is powerful and hence complex. It does contain duplication of features in order to satisfy differing approaches to how those features should be made available.

It should be noted that this requirement seems incompatible with

the others; to satisfy all the other TINMAN requirements will require a powerful - and not a simple - language.

For needed changes, note the following: there is no way that COBOL could be changed to satisfy this requirement.

I6.COBOL.Translator Restrictions
Fails to Satisfy

COBOL does not supply these limits except in a few very special cases. There are a number of items defined by the implementer, and in fact, a list is given on pages I-7 through I-8. However, most of these items apply to naming of certain things such as the computer name in the ENVIRONMENT DIVISION.

For needed changes, note the following: it would not be any more difficult to specify these restrictions for COBOL than for any other language.

I7.COBOL.Object Machine Restrictions
T (Satisfies)

COBOL does not contain language restrictions which are dependent on the object environment. It is not possible to comment on this for COBOL translators in general.

J1.COBOL.Efficient Object Code
Partially Satisfies

COBOL has an enormous number of optional features which the programmer can include or exclude in his source program. Presumably any good translator will not produce object code for these optional features when they are not used in the program. The input/output specifications assume that the interface with the hardware is being handled outside the translator. There is no automatic movement of programs between main and backing store except under programmer control via the Segmentation Module (section IX). The Segmentation Language Facility allows the user to indicate what is to be fixed in the memory at object time, and what can be overlaid.

For needed changes, note the following: we cannot comment on the required modifications without detailed study of many translators.

I2.COBOL.Optimizations Do Not Change Program Effect
Unknown

This is a requirement on translators and cannot be answered from the language specification.

J3.COBOL.Machine Language Insertions
Partially Satisfies

COBOL's access to machine-dependent hardware facilities is through ENVIRONMENT DIVISION, and so that portion of the requirement is met.

COBOL encapsulates the access and appearance of machine language code insertions through the use of the ENTER verb (page II-63). The machine language statements can either be written "in line" after the ENTER verb or elsewhere, but will be executed as if they had been compiled immediately following the ENTER statement. So COBOL satisfies the basic encapsulation requirement. It is unclear whether the requirement of "not be allowed interspersed with executable statement of the source language" is satisfied by this, but the most reasonable/likely interpretation would lead to the conclusion that COBOL does satisfy it. However, COBOL certainly fails the requirement that these insertions be permitted only within the body of the compile time conditional statements since there are no compile time conditional statements.

For needed changes, note the following: there would be no difficulty in satisfying the part of this requirement that involves inclusion only in compile time conditional statements if they existed (per requirement I4). However, this part of requirement J3 seems to defeat the purpose of allowing machine language instructions in the source program at all.

J4.COBOL.Object Representation Specifications
T (Satisfies)

The Data Description (page I-119) permits the user to specify the object representation of data. The order of fields is shown by the simple sequencing and level structure of COBOL (page II-17), the width of fields is shown via the PICTURE clause (pages II-18 through II-26); the presence of "don't care" fields is shown by using the word FILLER (page II-15) and the position of word boundaries can be controlled by the SYNCHRONIZED clause (pages II-33 through II-34) The user can also specify JUSTification of non-numeric data items (page II-16). However, it is not possible within COBOL to associate source language identifiers

with special machine addresses (to do so certainly would limit portability).

In connection with the text requirement that "object data specifications should be used sparingly", it is important to note that since COBOL has as its objective the efficient handling of large masses of data and data files, it is expected that the user will include the object data information in all cases. However, that would not be necessary for cases in which the data representation was not very important. It seems impossible to judge what language features are "spartan" and what ones are "grandiose".

The primary user control for positioning of data with respect to word boundaries is the SYNCHRONIZED clause (pages II-33 through II-34). If the user does not specify SYNCHRONIZE LEFT or RIGHT, then the translator will determine the most effective positioning of the data item.

J5.COBOL.Open and Closed Routine Calls
Fails to Satisfy

COBOL does not provide this capability in any form.

For needed changes, note the following: it would be relatively easy to add this capability. However, it should be pointed out that this seems an extremely undesirable requirement. While it is useful as long as a program is running on a specific machine with a known translator, its use can hamper efficient portability. Thus, the programmer might specify an open subroutine for a particular configuration and translator and yet when that program is moved to another machine and/or translator, it might be far more efficient to have a closed subroutine. Since the programmer has already specified which it is to be, the translator is unable to make the determination. It would be far better to leave this decision entirely in the hands of the translator.

K1.COBOL.Operating System Not Required
T (Satisfies)

While most COBOL implementations assume that there is an operating system, there is nothing in the language specifications requiring one. When there is an operating system, the ENVIRONMENT DIVISION and the USE verb provide the appropriate interface between the hardware/operating system combination.

K2.COBOL.Program Assembly
I (Satisfies)

The Inter-Program Communication module of COBOL (Section XII) does allow the creation of separately written modules for both type definitions and executable routines. The COPY statement (pages X-2 through X-4) serves the purpose of incorporating text into a COBOL source program. The COPY statement can be used in any sensible place within a source program (page X-2, point 7). COBOL does not have a language requirement specifying the form of the separately defined program modules.

K3.COBOL.Software Development Tools
Partially Satisfies

COBOL contains a Debug Module which gives the user control over debugging facilities; at compile time, the user is able to specify whether these should be translated for use at object time. Some translators contain the desired tools.

For needed changes, note the following: there are no restrictions in COBOL which would prevent the interface with software tools that might be developed. Support packages could easily be added to the language itself if the requirements were defined.

K4.COBOL.Translator Options
Partially Satisfies

COBOL does not itself contain any options to aid the generation, test, or modification of programs. COBOL does aid documentation by insisting on a particular reference format, and by the inherent readability which is a fundamental aspect of COBOL. Some translators supply these tools but it is impossible to specify exactly what all translators do.

For needed changes, note the following: it should be easy to include additional specifications along these lines once the requirements were determined.

K5.COBOL.Assertions and Other Optional Specifications
Partially Satisfies

COBOL contains a whole Debug Module (Section XI) but does not currently permit inclusions of assertions, assumptions, etc.

For needed changes, note the following: it should not be too difficult to include the missing language elements.

L1.COBOL.No Superset Implementations
Fails to Satisfy

The COBOL standard does permit an implementer to include language features which are not in the ANSI COBOL language "even though...it may prevent proper compilation of some programs that meet the requirements of this standard" (page I-6).

For needed changes, note the following: it would be trivial to change this rule to meet the requirement of L1.

L2.COBOL.No Subset Implementation
Fails to Satisfy

COBOL does not meet this requirement because the standard was specifically designed to permit a modular approach to implementation. Thus, there is a Nucleus and 11 specifically defined language modules (page I-1, section 1.2). Furthermore, many of the modules have subsets and the implementer is essentially able to choose what he wishes.

For needed changes, note the following: obviously it would be trivial to change the current definition of the standard to meet any particular subsetting requirement of TINMAN. There is no problem in the language specification.

L3.COBOL.Low-Cost Translation
Partially Satisfies

There are a great many options available to the COBOL programmer and his use of all of these will obviously increase compile time costs. Conversely, his failure to use many of them will decrease the compile time costs. There are no obvious cases in which the programmer can directly control the trade-off between compile-time and run-time costs. By now there are many efficient compilers of COBOL and hence the language is capable of being compiled fairly efficiently. Nothing can be said about translators for a future language.

For needed changes, note the following: it is not possible to state which changes should be made to allow COBOL to fully satisfy this requirement.

L4.COBOL.Many Object Machines
T (Satisfies)

COBOL 68 has been implemented for almost all full size computers and even one microprocessor. Many features in COBOL 74 were also included the later COBOL 68 implementations. The OBJECT COMPUTER paragraph (pages II-6 through II-7) in the ENVIRONMENT DIVISION makes this specification quite easy for the user.

L5.COBOL.Self-Hosting Not Required
T (Satisfies)

The CONFIGURATION Section in the ENVIRONMENT DIVISION (pages II-5 through II-7) enables the user to specify both the source computer on which the translation will be done and the object computer on which the program is to run. There are COBOL 68 compilers for most computers.

L6.COBOL.Translator Checking Required
Unknown

This is not a language requirement.

L7.COBOL.Diagnostic Messages
Fails to Satisfy

COBOL does not contain a suggested set of error and warning situations for the translator to implement.

For needed changes, note the following: it would be relatively easy to add such suggestions to the language definition.

L8.COBOL.Translator Internal Structure
T (Satisfies)

COBOL does not dictate to a translator the structure or characteristics of a particular translator.

L9.COBOL.Self-Implementable Language
T (Satisfies)

COBOL compilers can be written in COBOL; a very early version of a COBOL compiler was written in COBOL.

M1.COBOL.Existing Language Features Only
T (Satisfies)

Clearly COBOL is currently composed of features within the state-of-the-art. Furthermore, any changes necessary to achieve the requirements of TINMAN are in the category of engineering and not research.

M2.COBOL.Unambiguous Definition
Fails to Satisfy

The semantics of COBOL have not been defined in any formal fashion and currently contain ambiguities as is true of any English language definition.

For needed changes, note the following: probably it would be very difficult, although not impossible, to prepare a formal definition of COBOL semantics.

M3.COBOL.Language Documentation Required
Partially Satisfies

The COBOL standard contains a complete technical description of the language. The syntax is defined formally but the semantics are not defined in a formal fashion. The COBOL standard also contains a fair amount of tutorial or introductory material, even though it is not labelled as such and is scattered through the standard.

The only places in which characteristics of the source language are dependent on the translator, are those characteristics labelled as implementer-defined (pages I-7 through I-8). The current language standard document does not contain examples of each language construct, although it does contain listings of key words and language defined defaults. The standard is well written, and is put together for use by knowledgeable people. It is easy to find specific information in it.

For needed changes, note the following: it would be fairly simple to write tutorials for inclusion as part of the official documentation. There exist books and specific manuals on COBOL which serve this purpose. Instead of adding examples of each language construct, it would be more appropriate to have implementers issue documents on

expensive and inexpensive features. These features tend to vary depending on the translator.

M4.COBOL.Control Agent Required
T (Satisfies)

This requirement refers to the future. However, from its very beginning, COBOL has been "configuration managed". The defining authority has been CODASYL, and its subcommittee has been defined as the group responsible for COBOL. The people developing the standard have long accepted the principle that the only thing that they can do is pick and choose from among the features in the CODASYL-defined version of COBOL. Further, the U.S. Navy and NBS have provided validation services. Thus, there is a lot of experience and precedent in this management concept for COBOL. It seems to have worked fairly well.

M5.COBOL.Support Agent Required
Unknown

This is not truly a language requirement but is a consideration for the future.

A strong standards and configuration management agency required by M4 could presumably perform compiler validation, tools design, library standards and distribution, etc.

M6.COBOL.Library Standards and Support Required
Unknown

This is not truly a language requirement but is a consideration for the future.

A strong standards and configuration management agency required by M4 could presumably perform compiler validation, tools design, library standards and distribution, etc.

## Section IVc - COBOL Language Features Not Needed

### #.COBOL.Features To Be Eliminated From the Language

Because of its great power in the data handling and input/output area, there are a number of features in COBOL 74 which could be eliminated as unnecessary for TINMAN requirements. Since the COBOL syntax contains many optional phrases within verbs and clauses, it is not practical to list each case which can be eliminated. Thus, the following represents only a summary of the general features which can be eliminated, based on the COBOL 74 ANSI Standard and the TINMAN requirements. This list is not necessarily complete; (i.e., there are other language elements which could be deleted but a truly detailed listing would serve no useful purpose):

IDENTIFICATION Division

Relative I/O Module

Indexed I/O Module

Sort-Merge Module

Report Writer Module

Various editing characters (e.g., credit, debit, zero suppress, check protect)

Figurative constants

Some of the class conditions

Abbreviated combined relation conditions

Specific verbs: ALTER, INSPECT, SEARCH, SET, STRING, UNSTRING

Miscellaneous options in many places.

The following table presents the TINMAN requirements by category and the number in each category which COBOL 74 fully, partially, or failed to satisfy, as well as those which are not relevant or applicable. It is not surprising to note that COBOL does best in areas A, B, D and F. They are, respectively, Data and Types; Operation; Variables, Literals and Constants; and Scopes and Libraries. COBOL is poorest in E (Definition Facilities) since COBOL has no definition facilities aside from subroutines and does rather poorly in Category C (Parameters and Expressions). The entire matter of structure of programs and parameter passage is certainly one of COBOL's weakest concepts.

| REQUIREMENT | | COBOL |
|---|---|---|
| A. | Data and Types | |
| A1. | Typed Language | T |
| A2. | Data Types | P |
| A3. | Precision | P |
| A4. | Fixed-Point Numbers | T |
| A5. | Character Data | T |
| A6. | Arrays | T |
| A7. | Records | T |
| B. | Operations | |
| B1. | Assignment and Reference | T |
| B2. | Equivalence | T |
| B3. | Relationals | T |
| B4. | Arithmetic Operations | T |
| B5. | Truncation and Rounding | T |
| B6. | Boolean Operations | P |
| B7. | Scalar Operations - On arrays | T |
| B8. | Type Conversion - Implicit | F |
| B9. | Type Conversion - Explicit | P |
| B10. | I/O Operations | T |
| B11. | Power Set Operations | F |
| C. | Expressions and Parameters | |
| C1. | Side Effects | T |
| C2. | Operand Structure | T |
| C3. | Expressions Permitted | F |
| C4. | Constant Expressions | F |
| C5. | Consistent Parameter Rules | P |
| C6. | Type Agreement in Parameters | F |
| C7. | Formal Parameter Kinds | F |
| C8. | Formal Parameter Specification | T |
| C9. | Variable Numbers of Parameters | F |
| D. | Variables, Literals and Constant | |

| | | |
|---|---|---|
| D1. | Constant Value Identifiers | T |
| D2. | Numeric Literals | T |
| D3. | Initial Values of Variables | T |
| D4. | Numeric Range and Step Size | T |
| D5. | Variable Types | T |
| D6. | Pointer Variables | F |
| E. | Definition Facilities | |
| E1. | User Definitions Possible | P |
| E2. | Consistent Use of Types | F |
| E3. | No Default Declarations | F |
| E4. | Can Extend Existing Operators | F |
| E5. | Type Definitions | F |
| E6. | Data Defining Mechanisms | F |
| E7. | No Free Union or Subset Types | T |
| E8. | Type Initialization | F |
| F. | Scope and Libraries | |
| F1. | Separate Allocation and Access Allowed | F |
| F2. | Limiting Access Scope | P |
| F3. | Compile Time Scope Determination | T |
| F4. | Libraries Available | T |
| F5. | Library Contents | T |
| F6. | Libraries and Compools Indistinguishable | T |
| F7. | Standard Library Definitions | P |
| G. | Control Structures | |
| G1. | Kinds of Control Structures | P |
| G2. | *The GOTO* | P |
| G3. | Conditional Control | P |
| G4. | Iterative Control | T |
| G5. | Routines | F |
| G6. | Parallel Processing | F |
| G7. | Exception Handling | T |
| G8. | Synchronization and Real-Time | P |
| H. | Syntax and Comment Conventions | |
| H1. | General Characteristics | P |
| H2. | No Syntax Extensions | T |
| H3. | Source Character Set | T |
| H4. | Identifiers and Literals | P |
| H5. | Lexical Units and Lines | F |
| H6. | Key Words | P |
| H7. | Comment Conventions | P |
| H8. | Unmatched Parentheses | T |
| H9. | Uniform Referent Notation | T |
| H10. | Consistency of Meaning | T |
| I. | Default, Conditional Compilation and Language Restrictions | |
| I1. | No Defaults in Program Logic | T |
| I2. | Object Representation Specifica- | T |

| | | |
|---|---|---|
| | tions Optional | |
| I3. | Compile Time Variables | T |
| I4. | Conditional Compilation | F |
| I5. | Simple Base Language | F |
| I6. | Translator Restrictions | F |
| I7. | Object Machine Restrictions | T |
| J. | Efficient Object Representations and Machine Dependencies | |
| J1. | Efficient Object Code | P |
| J2. | Optimizations Do Not Change Program Effect | U |
| J3. | Machine Language Insertions | P |
| J4. | Object Representation Specification | T |
| J5. | Open and Closed Routine Calls | F |
| K. | Program Environment | |
| K1. | Operating System Not Required | T |
| K2. | Program Assembly | T |
| K3. | Software Development Tools | P |
| K4. | Translator Options | P |
| K5. | Assertions and Other Optional Specifications | P |
| L. | Translators | |
| L1. | No Superset Implementations | F |
| L2. | No Subset Implementations | F |
| L3. | Low-Cost Translation | P |
| L4. | Many Object Machines | T |
| L5. | Self-Hosting Not Required | T |
| L6. | Translator Checking Required | U |
| L7. | Diagnostic Messages | F |
| L8. | Translator Internal Structure | T |
| L9. | Self-Implementable Language | T |
| M. | Language Definition, Standards and Control | |
| M1. | Existing Language Features Only | T |
| M2. | Unambiguous Definition | F |
| M3. | Language Documentation Required | P |
| M4. | Control Agent Required | T |
| M5. | Support Agent Required | U |
| M6. | Library Standards and Support Required | U |
| Totals | T (Fully Satisfies) | 46 |
| | P (Partially Satisfies) | 22 |
| | F (Fails to Satisfy) | 26 |
| | U (Unknown) | 4 |

Although COBOL is widely used and implemented for its intended class of applications, THERE WOULD PROBABLY BE GREAT PSYCHOLOGICAL RESISTANCE TO ITS ADOPTION AS A BASIS FOR TINMAN WHATEVER COBOL'S TECHNICAL MERITS MIGHT BE WITH RESPECT TO OTHER LANGUAGES.

COBOL 74 considered as a base language for the DoD HOL is:

a. Moderately compliant with existing TINMAN requirements

b. Easily changed to meet approximately 23 of the requirements it does not fully meet

c. Moderately easily changed in approximately 9 of the requirements it does not fully meet

d. Difficult or impossible to change in approximately 13 of the requirements it does not fully meet

e. A viable base technically, but is probably unacceptable psychologically because of its verbosity and past history and the low opinion of it held by language experts.

Note that the numbers above represent complete compliance and include some requirements rated "90%, fully satisfies" in the report.

## Section V - HAL/S EVALUATION

### Section Va - HAL/S Language Introduction

#### #.HALS.Introduction

HAL/S is a higher order programming language developed by Intermetrics, Inc. for the flight software of the NASA Space Shuttle program. The language is expressly designed to allow programmers, analysts, and engineers to create software which is reliable, efficient, highly readable, and easily maintained.

HAL/S is intended to satisfy virtually all the flight software requirements of the Space Shuttle. To achieve this, the language incorporates a very wide range of features, including applications oriented data types and computations, real-time control, and constructs for implementing systems programming algorithms.

#### Data Types and Computations

HAL/S is a linear algebraic language particularly suited for aerospace proramming. Integer, scalar, vector and matrix data types, together with appropriate operators and built-in functions (e.g., a complete vector/matrix library) provide an extremely powerful tool for the implementation of mathematical (navigation, guidance and control) algorithms. Bit and character string processing constructs are available. The formation and use of multidimensional arrays, and of tree-like organizations of heterogeneous data, are also featured.

#### Real-Time Control

HAL/S is a real-time control language. With time being a critical parameter in many on-board solutions, time dependence has been introduced into HAL/S as part of the syntax. A wide range of commands for controlling real-time tasks is provided including one-shot and cyclic scheduling based on time, priority and hardware and/or software events.

#### Program Reliability

A major goal of the HAL/S design is the production of reliable software. Effective isolation between separately compiled program blocks contributes to modularity while communication is supervised

through centrally managed and highly visible common subroutines and data pools. Access to common resources may centrally be granted or restricted. For a real-time environment, HAL/S includes a locking mechanism which can automatically protect shared data and code.

Error Recovery

HAL/S has a comprehensive and flexible mechanism for detecting and recovering for run-time errors. It also has the capability of simulating run-time errors, which can be extremely useful for checkout purposes. Another feature of the language is the ability to specify and signal user-defined run-time errors.

Every active real-time process possesses its own so-called "error environment", which is essentially a description of the recovery actions in force for all possible run-time errors the process could be subject to. On initiation of the process, the system recovery action is in force for all run-time errors. During the life of a process, its error environment may be modified by the specification of a "user recovery action" for some error or error group. The user recovery action is enforced by the execution of specific HAL/S error control statements.

Systems Programming Features

HAL/S contains a number of features especially designed to facilitate its application to systems programming, thus substantially eliminating the necessity of using an assembler language. Two of these features are the facility to create and manipulate data pointers and to create language extensions in the form of "%macros".

a. Data Pointers - In HAL/S, pointer data items are called "NAME" data items. To substantially eliminate software unreliability, a specific mechanism in HAL/S assures that any given NAME data item can only point to other data items of a single kind specified at compile time. The mechanism consists in declaring a NAME data item with properties of type, precision, and arrayness, just as if it were an ordinary data item. These properties, rather than actually belonging to the NAME data item, are the properties which must be processed by data items to which the NAME data item can point.

b. %Macros - Realizing that it may be necessary to provide capabilities not basic to the language, HAL/S isolates these "system extensions" to a set of compiler-supplied %macros. These are written in assembler language and may be designed to meet specific requirements and interfaces. %Macros are

incorporated in the compiler run-time library or inserted as in-line code. As with other compiler "built ins", they benefit from the automatic checking conducted at compile time to assure proper usage. In effect, the %macro provides a means of adding functional, special-purpose implementation-dependent extensions to HAL/S without requiring far reaching syntax changes or substantial modification of the compiler programs.

Below are listed some of the strong points of HAL/S:

1. Real-time features include multitasking, event processing, and shared data management.

2. Built-in data types and operators for vector and matrix data are useful in programming spacecraft guidance and control systems.

3. Structured output listings with indentation for nesting levels are very readable and are somewhat self-documenting.

4. It is a block-oriented language with standard name scoping features. Separately compiled program blocks can be executed together and can communicate through one or more highly visible data pools (compools), which are themselves separately compilable.

5. Reliability is enhanced through strong type checking and structured programming constructs.

6. An error recovery facility allows the programmer freedom to define his own error processing logic or leave control with the system.

Some of the weak points of HAL/S:

1. There is a lack of "open-ended" capabilities - no extensibility, no generic procedures, and no recursion. It shall be noted that these are intentional omissions because of time and space restrictions inherent in an embedded, man-critical system.

2. There is a basic lack of versatility in the I/O capabilities.

3. No specific symbol is used for multiplication; the adjacency (with a space) of two operands indicates multiplication.

Section Vb - HAL/S Comparative Evaluation

Following is an itemized comparison of HAL/S against the DoD TINMAN requirements for a High-Order Language (HOL). An additional paragraph gives a brief discussion of the source of modifications required to fulfill TINMAN requirements. The reference document used is the "HAL/S Language Specification", Intermetrics Inc., Version IR-61-8, June 16, 1976. Each major section is prefaced with general comments and a summary.

A.HALS.Data and Types

HAL/S is a strongly typed language with a diverse set of data types. All data in a HAL/S program is defined by DECLARE statements. The group of all DECLARE statements valid for a program scope appear at the beginning of that scope in the data declaration block.

HAL/S adequately fulfills the TINMAN requirements for a strongly typed language, with two exceptions. There is no fixed-point capability and character sets are not treated as enumeration types. Additional capabilities include the EVENT, BIT, NAME, MATRIX, and VECTOR data types - each with their own set of characteristics and operation.

A1.HALS.Typed Language
T (Satisfies)

HAL/S requires all data to be declared and fully qualified in the source program (Section 4.0, page 4-1 through Section 4.7, page 4-22). There exists a set of default properties for all data items (e.g., floating-point, single precision, non-initialized) which HAL/S assumes if none is given (see Lang. Spec. page 4-19). But the data identifier must appear in a declaration statement for this to occur. HAL/S does not allow dynamic declaration.

A2.HALS.Data Types
Partially Satisfies

HAL/S provides integer (INTEGER), real-floating-point (SCALAR), Boolean (BOOLEAN), and character (CHARACTER) basic data types. Five additional types are also provided: BIT, VECTOR, MATRIX, NAME, and EVENT (Section 4.7, page 4-19). The BIT designation is simply a string of Booleans. The VECTOR and MATRIX designation defines a specified (or, if unspecified, a default) number of floating-point values which are naturally related. The NAME is a pointer variable which can be manipulated. The EVENT designation is a specially defined Boolean with

real-time programming implications. All HAL/S data items listed above may be defined in groups of homogeneous types called ARRAYs (Section 4.5). All data items or arrays of data items may be defined in groups (copies) of generally non-homogeneous types called STRUCTURE's (Section 4.3). Indexing to the STRUCTURE copy, ARRAY element and, in the case of CHARACTER, BIT, MATRIX, and VECTOR, to the component level, is provided.

Modification required: modification of HAL/S to provide fixed-point capability is moderately difficult. HAL/S has available a simple fixed-point capability which is currently being used on some applications. This facility is designed for machines which have no floating-point hardware. Although currently there is no implementation of HAL/S which has both fixed-point and floating-point capabilities, it is feasible to have such. The compiler modifications (library routines, additional type checks, etc.) to provide such a combination of capabilities would be moderate. This evaluation has deemphasized the presence of this additional feature in analysis of individual requirements dealing with fixed-point capabilties.

A3.HALS.Precision
T (Satisfies)

HAL/S does not provide the capability to specify or change precision globally (on the scope level). It does permit precision specification for individual variables at DECLARE time (Section 4.7, page 4-20, general rules 2 and 3). All floating-point and integer arithmetic is done in single precision unless one or more of the operands have been explicitly declared to be double precision or unless an explicit conversion is requested in the expression (Section 6.1.1, page 6-4).

A4.HALS.Fixed-Point Numbers
Fails to Satisfy

HAL/S, as specified in the reference documents, has no fixed-point capability.

Modification required: modification of HAL/S to provide fixed-point capability is moderately difficult. HAL/S has available a simple fixed-point capability which is currently being used on some applications. This capability is designed for machines which have no floating-point hardware. Although currently there is no implementation of HAL/S which has both fixed- and floating-point capabilities, it is feasible to have such. The compiler modifications (library routines, additional type checks, etc.) to provide such a combination of capabilities would be moderate. This evaluation has deemphasized the

presence of this additional feature in analysis of individual requirements dealing with fixed-point capabilties.

A5. HALS.Character Data
Fails to Satisfy

HAL/S does not consider character strings as enumeration types, nor does it provide the capability to specify an "order of characters" at compile time. A single character set is defined for the language. This character set is in itself an enumeration type, with an implied collating sequence and subject to all relational operators (see B3).

Modifications required: to add the capability for sets of enumerated types and to include character sets would be a difficult modification.

A6.HALS.Arrays
T (Satisfies)

HAL/S provides for up to n dimensions (where n is implementation-dependent) in ARRAY declarations (Section 4.5, page 4-14, rule 2). The dimension number(s) specify the upper bound of the range of each dimension. (The lower bound is understood to be 1.) The dimension value may be expressed as an arithmetic expression, but is rounded to the nearest integer value. The uppermost bound of an array which appears as a parameter of a Procedure or Function may be determined at entry into the Procedure/Function, for one dimensional arrays only (Section 4.5, page 4-16, restriction #1).

A7. HALS.Records
T (Satisfies)

The STRUCTURE facility in HAL/S fulfills this requirement (Section 4.3). It provides for a hierarchical organization of generally non-homogeneous data types. The NAME facility (described under TINMAN D6) applied to structures, provides an "overlay" capability with type checking.

B. HALS.Operations

HAL/S has three basic categories of operators (corresponding to three categories of data types): arithmetic, Boolean, and character. The largest number of operators fall into the arithmetic category. HAL/S uses a standard symbol with infix notation for all operations.

HAL/S adequately fulfills all requirements to provide operations to be used by the data types described in section A. The single exception is the requirement for operations on "power sets of enumeration types". Other weaknesses in the operation category include implicit rounding and truncation in some cases, implicit type conversion in some cases, and no dynamic I/O device assignment. HAL/S exceeds TINMAN operation requirements in that it provides built-in operations for vectors and matrices.

B1.HALS.Assignment and Reference
T (Satisfies)

HAL/S provides automatic assignment operations as a part of the definition for all data types. Any variable data item may have the value of another variable, a constant value, or the result of an expression assigned into it, providing the data types are equivalent. There are, however, a few exceptions to data type equivalency (Section 7.3, page 7-6, general rule 5). The uniform symbol for assignment for all data types is the "=" sign (Section 7.3). Composite data types (e.g., arrays, structures, matrices) may be treated as a single entity in assignment or reference (Section 7.3, page 7-5):

```
ARRAY1 = ARRAY2 + ARRAY3;
```

Additionally, HAL/S has a multiple-assignment capability which allows the value of a single operand to be stored into several variables in one statement (Section 7.3, page 7-5):

```
FIRST, I, LOOP-COUNTER = 1;
```

B2.HALS.Equivalence
T (Satisfies)

The symbol for identity or logical equivalence is the "=" sign. (See comments in H10.) Equivalence comparisons must be between like data types. Arithmetic expressions cannot be implicitly compared to Booleans; nor character strings to arrays, etc. However, mixed expressions of integer and floating-point operands are accepted for arithmetic equivalence comparison (Section 6.2.1). Mixed precision is also accepted within arithmetic comparisons. HAL/S also has equivalence comparison capability between aggregate data items such as arrays, structures, vectors, and matrices. An array can be compared to a single item (of the same type) for identity. (Identity exists only if all elements of the arrayed operand are equal to the unarrayed operand.) An array may also be compared to another array of identical dimension for equivalnce (Section 6.2.5). Evaluation of equivalence for each

corresponding pair of elements of the operands holds true in comparing vectors, matrices, character strings, Boolean strings and structures (Section 6.2.1, rule 4; Section 6.2.2, rule 3; Section 6.2.3, rules 1 and 4; Section 6.2.4, rule 4).

B3.HALS.Relationals
T (Satisfies)

All arithmetic data types in HAL/S have defined for them six basic relational operators ("=", " =", ">", ">=", "<", "<="). The [algebraic not sign] is the symbol for NOT. Alternatively the keyword "NOT" may be used in conjunction with "=", "<", ">". The relational operators "=" and "NOT=" may also be applied to the VECTOR, MATRIX, BOOLEAN, ARRAY, and STRUCTURE data (Section 6.2.1).

B4.HALS.Arithmetic Operations
Partially Satisfies

HAL/S has the set of arithmetic operations listed, although it uses a space or blank instead of a symbol for multiplication (Section 6.1.1, page 6-3). Integer division with remainder is not provided in the set of infix operators. The user can obtain the results of integer division by way of two built-in functions. The function DIV (alpha, beta) (where alpha and beta are integers) yields an integer result with no remainder (Appendix C, page C-1). The function REMAINDER (alpha, beta) (where alpha and beta are integers) yields the integer remainder resulting from the division of beta into alpha (Appendix C, page C-1).

Standard multiplication is provided for all data types, including vectors and matrices. Additionally, HAL/S provides a dot product and a cross product operation for vectors (Section 6.1.1, page 6-4). Operations involving vectors or matrices compile into machine code which does the specified operation iteratively on each element. However, this provides the programmer a cleaner listing and often times more concise machine code.

Modifications required: the capability to provide integer division with remainder as an infix operator could be provided with no difficulty.

B5.HALS.Truncation and Rounding
Partially Satisfies

A certain degree of implicit rounding or truncation takes place in

assignment operations between arithmetic values of different types or precision (Section 7.3, page 7-6, semantic rules):

1. Assignment of an integer expression into a floating-point variable results in an implicit conversion with a possible loss of decimal places of accuracy.

2. Assignment of a floating expression into an integer variable results in rounding to the nearest integral value. This may cause an execution exception if the absolute value is too large to be represented as an integer.

3. Assignment of a double precision floating-point value into a single precision floating-point variable will result in truncation of binary digits from the mantissa.

4. Assignment of a double precision integer value into a single precision integer value will result in truncation of the high-order digits. (Only the low order bits are stored.)

Modifications required: the default rounding, truncation and implicit conversion could be eliminated, but not without some difficulty (moderate). Explicit calls to conversion routines could easily be replaced with calls to diagnostic routines. However, many implicit conversions peculiar to the object machine would have to be anticipated and prefaced with calls to diagnostic routines.

B6.HALS.Boolean Operations
T (Satisfies)

HAL/S Boolean operators include "AND", "OR", and "CAT" (Section 6.1.2, page 6-7). The CAT operator is used to concatenate strings of Booleans. The logical complement of a Boolean expression may be obtained by prefacing it with "NOT" (Section 6.1.2, page 6-9). There is no infix operator for exclusive OR but a built-in function XOR (alpha, beta) (where alpha and beta are bit strings) provides this capability (Appendix C, page C-8). The "short circuit" mode exists for all connective Boolean operations between relational expressions. (Reference 6.1.2)

B7.HALS.Scalar Operations
T (Satisfies)

HAL/S permits assignment and operations on arrays (Section 6.1.5;

Section 7.3, page 7-5, rule 3) and structures (Section 6.1.4; Section 7.3, page 7-8) (records) of identical format. The operations only appear to provide a parallel operation on all elements simultaneously, while the machine code produced may perform an element-by-element operation.

B8.HALS.Type Conversion
Fails to Satisfy

HAL/S allows many implicit type conversions (Section 7.3, page 7-6 and 7-7; Appendix D).

Explicit conversion between all data types is provided for in HAL/S. For conversion to bit strings from character strings and vice versa, the introduction of a radix is allowed (Section 6.5.2, page 6-32; 6.5.3, page 6-34). This radix causes the conversion to be done in a specified number base:

```
CHARACTER[@DEC] (BIN'101101') = '45'   where [@] indicates a
                                       subscript
```

An additional capability is the SUBBIT pseudo-variable (Section 6.5.4) which allows access to bit representations within other data types:

```
SUBBIT[33 to 64] (DP)     where DP is a double precision
                          floating-point value, yields bits 33
                          through 64 of the machine representation
                          of DP and makes it look like a 32-bit
                          variable.
```

HAL/S also provides explicit precision conversion between floating-point and integer values (Section 6.6).

Modifications required: the default rounding, truncation and implicit conversion could be eliminated, but not without some difficulty. Explicit calls to conversion routines could be replaced with calls to diagnostic routines. However, many implicit conversions peculiar to the object machine would have to be anticipated and prefaced with calls to diagnostic routines.

B9.HALS.Changes in Numeric Representation
Partially Satisfies

All constant values are checked at compile time for a machine dependent maximum. HAL/S has no compile time warning for possible range errors within expressions, but does have at run time. Explicit conversion of a double-precision integer to single precision will not result in a run time exception. See B5 for other comments on possible error conditions connected with implicit type and precision conversion. (Reference Appendix D)

Modifications required: the addition of a fixed-point capability (see comments under A2) which uses range specification at declare time could certainly include compiler checks for range errors.

B10.HALS.Input/Output Operation
Partially Satisfies

HAL/S provides capabilities for sequential and random access I/O (Section 10.1; 10.2). The sequential I/O commands may be accompanied by control commands which cause explicit movement or positioning of device mechanism (Section 10.1.3). For specific I/O device interface the "%macro" capability may be used. (See comments on J3.) There is no operation which allows the program to dynamically assign or reassign I/O devices.

Modifications required: to provide all of the I/O capabilitis required, as a part of the standard language, would be difficult.

B11.HALS.Power Set Operations
Fails to Satisfy

HAL/S has no data types defined as power sets of enumeration types.

Modifications required: HAL/S would need to provide a subrange capability for enumeration types, allow these sets to be mapped to bit strings and provide special operations to manipulate them. The operations are in place for standard bit strings, but the capability would have to be expanded to encompass all enumeration types. This language change would be extremely difficult.

C.HALS.Expressions and Parameters

HAL/S has a set of rules that governs combining operands and operators to form expressions. The user (or reader) can easily determine the meaning of any expression from the written statement.

Also, there are strict rules for dealing with formal parameters of procedures and functions (e.g., agreement of type and number).

The requirements for expressions and parameters are basically fulfilled by HAL/S. There is no "generic procedure" capability, but this is a debatable requirement. If the intent of this requirement is to handle arrayed arguments or lists of output data, HAL/S adequately provides for these through built-in functions and standard I/O statements.

C1.HALS.Side Effects
T (Satisfies)

HAL/S has a definite precedence and evaluation order for operations in a single expression (Section 6.1.1, page 6-5). The rule of left-to-right evaluation is overridden for the multiplication operators when necessary to summarize the total number of elemental multiplications required (page 6-5, rule 3). However, the user may explicitly alter the hierarchy of evaluation order through use of parentheses.

C2.HALS.Operand Structure
T (Satisfies)

Operators and operands are clearly distinguishable in HAL/S expressions, except for the arithmetic multiply operator which is denoted by a space. (See comments on H10.)

HAL/S has six levels of operator hierarchy. The insertion of two vector operators causes what might be considered a large table of precedence. The levels of precedence are (Reference 6.1.1):

| | | |
|---|---|---|
| 1. | exponentiation | ** |
| 2. | multiplication | blank |
| 3. | cross-product | * |
| 4. | dot-product | . |
| 5. | division | / |
| 6. | addition | + |
| 7. | subtraction, negation | - |

C3.HALS.Expressions Permitted
T (Satisfies)

HAL/S allows expressions of any resultant type to appear in any

context in which a variable or constant of that type might appear (Section 6.1, page 6-6; Section 5.3.2, page 5-11). In particular, arithmetic expressions or normal functions - either "built-in" or user-defined - may be used as subscripts (see diagrams on page 5-11 and 6-6; Section 6.4, pages 6-23).

C4.HALS.Constant Expressions
T (Satisfies)

HAL/S evaluates all constant expressions at compile time (Appendix F).

C5.HALS.Consistent Parameter Rules
T (Satisfies)

All parameters of a specific type are treated consistently.

C6.HALS.Type Agreement in Parameters
T (Satisfies)

HAL/S does require arrayed formal parameters to have a specified number of dimensions. However, only single-dimension arrays may have the size of their dimension determined at entry into the array scope (Section 4.5, page 4-16).

C7.HALS.Formal Parameters
T (Satisfies)

HAL/S adequately fulfills the intent of this requirement, although there is no "class distinction" made between formal parameters. A formal class of parameters for specifying control action for exceptions does not exist. (See comments on G7.) Actual procedure parameters, although, not a separate "class", must be declared initially in the procedure and must agree in type and attributes with those of the calling arguments (Section 3.7.2, 3.7.3).

C8.HALS.Formal Parameter Specification
Fails to Satisfy

HAL/S requires the re-declaration of all formal parameters with their associated attributes in the procedure or function body. Generic procedures are not possible in HAL/S. Compile time type checks are performed on all formal parameters (Section 3.7.2).

Modifications required: see comments under C3.

C3.HALS.Variable Number of Parameters
Fails to Satisfy

HAL/S will not allow a variable number of input arguments to procedures or functions (Section 3.7.2).

Modifications required: certainly the concept of allowing a variable number of arguments on procedure calls and optional format parameter definition within the procedure itself is feasible. However, the compiler modifications necessary to implement this capability would be extremely difficult to make.

D.HALS.Variables, Literals, and Constants

HAL/S accomplishes the need to associate initial values with all data types. With the single exception of fixed-point range and step size specification, HAL/S meets or surpasses all requirements in this area.

For the modifications required to fully comply with TINMAN, see comments under section A for fixed-point discussion.

D1.HALS.Constant Value Identifiers
T (Satisfies)

HAL/S provides the capability to associate constant values with identifiers and has compiler checks preventing changes to them. This is specified with the CONSTANT attribute (Section 4.8).

D2.HALS.Numeric Literals
Unknown

HAL/S' compliance with this requirement is unknown from defined references.

D3.HALS.Initial Value of Variables
T (Satisfies)

The INITIAL capability may be used with any declared identifier

(Section 4.8), but if not used, generally no default value is assigned to it. The exceptions to this are the NAME pointer which defaults to 'null' (Section 11.4.10, page 11-34), and the EVENT variable which defaults to FALSE (Section 4.8, page 4-26).

The INITIAL capability may be used in connection with the keywords AUTOMATIC or STATIC (Section 4.5, page 4-14). The AUTOMATIC attribute causes an identifier with the initialization attribute to be initialized on every entry into the code block containing its declaration. The STATIC attribute causes an identifier to be initialized only on the first entry into the code block. Thereafter its value is guaranteed to be preserved for the next entry into the block. If neither attribute is specified, then STATIC is assumed.

D4.HALS.Numeric Range and Step Size
Fails to Satisfy

HAL/S does not have the capability to specify range and step size in its limited fixed-point feature. (See comments under A4.)

D5.HALS.Variable Types
Partially Satisfies

Any built-in data type may be associated with a variable or array (Section 4.5, page 4-13; Section 4.7, page 4-19). Components of structures may be structures or arrays (Section 4.3, page 4-10; Section 4.5). Structures may be multicopied, which gives them an array-like appearance (page 4-22). HAL/S has no user-defined types or subsequences of enumeration types to which this requirement could be applied (see comments in section E).

D6.HALS.Pointer Variables
T (Satisfies)

The NAME facility is a declared attribute which provides the pointer capability. To greatly eliminate software unreliability sometimes inherent in a pointer mechanism, HAL/S requires that NAME data items can only point to other non-pointer data items of a specific type defined at compile time. This is accomplished by declaring a NAME data item with properties of type, precision and "arrayness" just as if it were an ordinary data item. These properties, rather than actually belonging to the NAME data item, are the properties which must be possessed by data items to which the NAME data item can point. Type checking is done on all NAME data item manipulations (Section 11.4, page 11-16).

E.HALS.Definition Facilities

HAL/S does not have an extensibility feature as such. In the sense that user-defined functions and structures comprise "extension", HAL/S does comply with the intent of this requirement. If one purpose of requiring extended data types and operations is to provide a capability for creating and manipulating vectors and matrices, HAL/S has these features built-in and has no need for extension in this area.

To make the HAL/S language extensible by allowing user definitions of data and operations would be very difficult. A need for true extensibility has not been seen so far in any usage of HAL/S. The nature of the compiler-builder and ease of modification of the compilers has provided all the flexibility needed.

E1.HALS.User Definitions Possible
Partially Satisfies

HAL/S has a Function capability which defines a "new" operation of the user's own creation. However, it is not directly associated with a particular data definition, nor can it be used as an infix operator. The function invocation may appear within an expression (see comments on H9) giving the same results that would have been obtained if a user-defined infix operator had been specified.

The capability to add machine-dependent extensions to the language is provided in the "%macro" (Section 11.2.2). The invocation of a %macro may cause the compiler to generate inline object code to be executed directly or it may generate linkages to external routines. In neither case does the %macro act as a new user-defined data type or infix operator. However, it does provide some of the flexibility demanded by language extension.

The user data definition facility in HAL/S is even less obvious. The structure definition (Section 4.3) and manipulation capabilities partially fulfill the intent of user-defined data (see A2, A7, B1, B2, and B3).

HAL/S also has a REPLACE capability which can be used to define an identifier text substitution which is to take place wherever the identifier is referenced (Section 4.2, page 4-2 through 4-7.1).

An example would be:

```
REPLACE INCR(X) BY "X=X+1";
```

Thus, a simple increment function has been defined for the name scope of the definitions.

To make the HAL/S language extensible by allowing user definitions of data and operations would be very difficult.

E2.HALS.Consistent Use of Types
Fails to Satisfy

Even assuming the "extension features" described under E1, HAL/S does not fulfill this requirement.

To make the HAL/S language extensible by allowing user definitions of data and operations would be very difficult.

E3.HALS.No Fault Declarations
Fails to Satisfy

HAL/S allows many default attributes in data declarations (e.g., 3-vector, 3x3 matrix, floating-point single precision, etc.) (Section 4.7, pages 4-19 and 4-20).

Modifications required: the default attributes provided by the compiler could be easily removed and a compile-time error condition flagged for all data not fully declared and qualified.

E4.HALS.Can Extend Existing Operators
Fails to Satisfy

HAL/S has no capability to extend the meaning of operators.

To make the HAL/S language extensible by allowing user definitions of data and operations would be very difficult.

E5.HALS.Type Definition
Fails to Satisfy

HAL/S has no capability for user data type definition.

To make the HAL/S language extensible by allowing user definitions of data and operations would be very difficult.

E6.HALS.Data Defining Mechanisms
Fails to Satisfy

HAL/S has no capability for user data type definitions.

To make the HAL/S languge extensible by allowing user definitions of data and operations would be very difficult.

E7.HALS.No Free Union or Subset Types
T (Satisfies)

This is a negative requirement. HAL/S fulfills this requirement by default since there are no user data type definitions.

E8.HALS.Type Initialization
Fails to Satisfy

HAL/S has no capability for user data type definitions.

To make the HAL/S language extensible by allowing user definitions of data and operations would be very difficult.

F.HALS.Scopes and Libraries

HAL/S completely fulfills the requirements for language scopes and libraries and modifications are unnecessary.

F1.HALS.Separate Allocation and Access Allowed
T (Satisfies)

The ACCESS attribute may be applied to declared data, causing implementation-dependent managerial restrictions on access to be placed on the variable when it is used in assignment contexts (Section 4-5, page 4-15). ACCESS may also be applied to a program block (procedure or function) (Section 3.7.1, page 3-14.1).

The UPDATE block is used to control the sharing of data by two or

more real-time processes (Section 3.4, page 3-8). The variables being controlled must have in their declaration the attribute LOCK (n) - where "n" indicated a "lock group" (Section 8.10). The update block is an explicitly delimited body of code whereby the interaction of code blocks referencing or modifying locked data may be controlled.

F2.HALS.Limiting Access Scope
T (Satisfies)

The ACCESS capability as applied to data, compools, programs, procedures, or functions fulfills this requirement. (See comments under F1.)

F3.HALS.Compile Time Scope Determination
T (Satisfies)

HAL/S follows a rigid set of rules for determining the scope of identifiers for reference (Section 3.8). Identifiers may be referenced only in their "name-scope" (the code block containing the declaration, and all code blocks nested therein).

F4.HALS.Libraries Available
T (Satisfies)

HAL/S provides the capability to build a program complex from several units of compilation (program, external procedure, external function, or compool) (page 3-1). These units may be compiled independently in sequential invocations of a compiler or they may be drawn from implementation-dependent libraries. Any compilation unit referencing another unit must be preceded by a block template describing the referenced unit (Section 3.1, page 3-3; and Section 3.6). These block templates may be contained in implementation libraries and included via compiler directive.

F5.HALS.Library Contents
T (Satisfies)

HAL/S' programs may call routines written in other languages. The invocation of this capability is noted by the NONHAL keyword followed by an integer number indicating the type of program unit to be accessed (Section 4.6). The NONHAL attribute attached to a function or procedure name indicates to the compiler that a predefined linkage mechanism should be set up for that unit. Current implementations use the NONHAL capability to interface with 360 Assembler and FORTRAN code blocks.

However, the capability is open-ended and could easily be expanded to other languages.

F6.HALS.Libraries and Compools Indistinguishable
T (Satisfies)

HAL/S does not preclude the generation of libraries which may include programs, procedures, functions or compools. The compool may contain data of all types and may be associated with any unit of compilation in order to share data (Section 3.5). The control and access to completion units within libraries is accomplished with the "block template" facility (Section 3.6).

F7.HALS.Standard Library Definitions
T (Satisfies)

HAL/S provides standard capabilities for sequential and random access I/O (Section 10.1 and 10.2). Several miscellaneous machine-independent built-in functions provide machine-dependent capabilities (e.g., CLOCKTIME - time of day, RUNTIME - real-time executive clock time, DATE, etc.) (Appendix C, page C-5).

G.HALS.Control Structures

HAL/S provides all of the basic control structures required for controlling and altering program flow. It does not provide true recursion. The Boolean expression conditional control structure is not required to be fully partitioned, nor are control variables required to be local to the loops.

G1.HALS.Kinds of Control Structures
Partially Satisfies

HAL/S does not have control mechanisms for recursive control. There is a set of primitive control structures which provide all the other desired control mechanisms (Section 7.2, 7.6.2 through 7.6.6, 7.7).

See comments under G5 for "modifications required".

G2.HALS.The GOTO
T (Satisfies)

HAL/S has a GOTO feature which causes a branch in execution to a labeled executable statement within the same name scope. The GOTO may not cause execution to branch into a DO...END group, or into or out of a code block (Section 7.7).

G3.HALS.Conditional Control
Partially Satisfies

HAL/S provides all three conditional control structures mentioned in this requirement: IF...THEN...ELSE (Section 7.2), discrete DO FOR (Section 7.6.4), and the DO CASE (Section 7.6.2).

HAL/S does not require the IF...THEN...ELSE or the DO CASE to be fully partitioned. If the ELSE clause is presented in an IF...THEN...ELSE construct, it may not be preceded by an embedded IF...THEN statement without an ELSE clause:

```
IF (condition)

THEN

     IF (condition)      embedded IF...THEN
     THEN                not allowed.

ELSE (statement)
```

This prohibits the occurrence of the "dangling ELSE" problem. The embedded IF...THEN can be used if enclosed in a DO...END group.

Modifications required: to force an ELSE clause to be present in every IF...THEN...ELSE and DO CASE construct would be a relatively simple compiler modification. These clauses are implied by the very nature of the control structures and are conceptually present, though not seen. Either a compiler diagnostic message for omission or insertion of an explicit "ELSE <null>;" would be possible to satisfy this requirement.

G4.HALS.Iterative Control
Partially Satisfies

Control variables are not required to be local to the iterative control structure. An optional capability to provide temporary integer

variables (loop control counters, etc.) within the control structure is present in the TEMPORARY data declarative attribute (Section 11.3).

HAL/S provides the capability to specify more than one terminating predicate:

DO FOR (iterative control) WHILE (condition); (Section 7.6.2)

or

DO FOR (iterative control) UNTIL (condition); (Section 7.6.3)

Modifications required: actually no compiler modification is required for this capability. The TEMPORARY feature fulfills it completely. However, the programmer has an option to use this feature or to use non-local variables for loop control. A simple compiler modification would be to interpret all "DO FOR" constructs to be "DO FOR TEMPORARY".

G5.HALS.Recursive Routines
Fails to Satisfy

True recursion (invoking a routine from within itself) or co-routine recursion (calling routine B from routine A, and B, in turn, calling A again), is not allowed in HAL/S (Programmers' Guide, page 1-5).

Modifications required: the mechanism to make HAL/S recursive is basically present in the form of the totally reentrant capability. Addition of run-time checks for extreme stack sizes (caused by possible infinite loops) could be made.

G6.HALS.Parallel Processing
T (Satisfies)

HAL/S has a facility for creating a multiprocessing job structure in a real-time programming environment. The Real-Time Executive (RTE) controls the execution of processes held in a process queue. Processes may be in one of four states in the process queue - active, wait, ready, stall. Depending on the implementation, it is possible for several processes to be active (in execution) simultaneously (Section 8.1, page 8-2; Programmers' Guide, pages 3-2 and 13-4). HAL/S contains statements which schedule processes (enter them into the process queue - Section

8.3), terminates them (removes them from the queue - Section 8.4 and 8.5), and otherwise directs the RTE in its controlling function. Several processes may be put into the active state simultaneously by using the SCHEDULE statement and specifying immediate activation (Diagram, page 8-4; items 7 and 8, page 8-6).

G7.HALS.Exception Handling
T (Satisfies)

HAL/S allows the user to change the prevailing error recovery action via the ON ERROR command (Section 9.1). When an error occurs, the action to be taken, either system- or user-defined, is determined as a result of the ON ERROR statement in force at the time. HAL/S has no provision for passing data parameters to a recovery point. However, the error recovery system has access to information such as statement number, run time and trace back through calling sequence, that might be considered "parameterized data" (Section 9.0, second paragraph).

G8.HALS.Synchronization and Real-Time
T (Satisfies)

The WAIT capability (Section 8.6) fulfills this requirement by causing delay in execution for a time delta or until a certain notice event has been set (process event or I/O). HAL/S allows programs to execute in pseudo-parallel via the SCHEDULE statement (Secton 8.3, diagram). This statement has provisions to delay beginning execution until an event has taken place, at a certain time, or after a specified time delay. Priority of execution and repetition rates may be specified. Termination criteria (e.g., after an event, or time) may also be stated in the SCHEDULE statement (Section 8.3).

H.HALS.Syntax and Comment Conventions

Basically HAL/S has a very powerful and versatile syntax. The input is "stream-oriented", that is, statements may begin anywhere on a line (or card) and may overflow onto succeeding lines or cards. Certain input columns are reserved for special control information (see H5).

H1. HAL.General Characteristics
T (Satisfies)

HAL/S is a free-format language with statements made up of literals (Section 2.3.3), user-defined variables (Section 2.3.2), language unique keywords (Section 2.3.1), and certain special characters (see table on page 2-4). Blanks may be interspersed between most of the elements

making up a source line (Section 2.5). HAL/S allows up to 32 characters in identifiers with break characters interspersed as desired to enhance readability. The specific statement delimiter used is the semicolon. No keyword abbreviations are allowed.

H2.HALS.No Syntax Extensions
T (Satisfies)

HAL/S does not allow the user to modify the syntax of the source language.

H3.HALS.Source Character Set
T (Satisfies)

Following is a list of the character set used in HAL/S. It is a subset of the standard ASCII and EBCDIC sets.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789
+-*./  &=<>#  $,:;'"()( %
(blank)
```

The only HAL/S feature inaccessible using the 64 character ASCII subset only is the "escape mechanism". This capability allows the user to introduce within a character literal, a character not included in the HAL/S character set (Section 2.3.3, page 2-10, rule 5). It is introduced with the "[cent]" which is not a part of the basic 64-character ASCII subset.

The character set is easily modified by substitution of acceptable characters on an implementation basis. This would allow use of an acceptable ASCII character for the "[cent]" sign.

H4.HALS.Identifiers and Literals
T (Satisfies)

Very precise syntactical rules for identifiers (Section 2.3.2) and literals (Section 2.3.3) are specified for HAL/S. The break character used is the underscore. The space or blank is usually a delimiter unless it is embedded in a character string or is meant to signify multiplication (see B4 and H10). The insertion of a break character within an identifier name causes a unique identifier to be created (e.g., ALPHA-BETA is not identical to ALPHABETA).

H5.HALS.Lexical Units and Lines
Partially Satisfies

HAL/S partially violates this requirement by allowing the continuation of character literals across input lines. All other lexical units must be contained on a single input line.

HAL/S allows a mixed single or multiline input format. The single line format allows the user to denote exponentiation (**) and subscripting ($) on the same line (Section 2.4). Thus, the expression

$$A = X^{(Y+2)} + B_{(Z-2)}$$

would be written A = X** (Y+2) + B$ (Z-2); in single line format. The multiline format allows the user to input the subscripts and exponents as they appear mathematically -- below and above the main line respectively. The first character position of each line is reserved for a symbol (E,M,S) denoting the exponent, main, or subscript lines. The previous example would be written for multiline input:

```
E        Y + 2
M   A = X      + B     ;
S                Z - 2
```

Modifications required: the capability to allow embedded comments and character strings to continue across input lines could be easily modified to allow concatenation of literal strings implicitly and thus eliminate the one case in which HAL/S violates this requirement.

H6.HALS.Key Words
T (Satisfies)

HAL/S has both keywords (Appendix B) and built-in function names (Appendix C) which are not available for any use other than that which is intended by their definition in the language specification. There are 95 keywords and 56 built-in function names for a total of 151 reserved words.

H7.HALS.Comment Conventions
Partially Satisfies

HAL/S allows embedded comments within a source statement (Section 2.5). The format is:

/* ... Any Text ... */

Such comments may appear between statements, but may not appear in a literal, reserved word, or identifier. The comment is terminated by a delimiter rather than the end of a source line. There are implementation-dependent restrictions on the overflow of embedded comments from line to line of the source text.

Additionally, HAL/S allows a single line comment which has no executable statements in it. A "C" in column one of the input line indicates the beginning of the comment; it is terminated at end-of-line (Programmers' Guide, page 2-11).

H8.HALS.Unmatched Parentheses
T (Satisfies)

No unmatched parentheses, brackets, or braces are allowed in HAL/S; nor are unbalanced DO...END pairs allowed.

H9.HALS.Uniform Notation
T (Satisfies)

HAL/S allows function invocation to occur anywhere that the resultant quantity returned from the function would be valid (Section 6.4).

H10.HALS.Consistency of Meaning
Fails to Satisfy

Depending on the context in which it appears, HAL/S uses the "=" sign to indicate either assignment or equality (Section 6.2.1, 7.3).

The blank character may also have different meanings, depending on the context in which it is used. It may be used as a symbol separator or a delimiter but may also be used to signify multiplication (Section 6.1.1). The adjacency of two operands separated by a blank indicates multiplication. This can sometimes be confusing to the user. Particularly misleading would be the case of inadvertently omitting another operator sign, resulting in a syntactically correct but logically incorrect expression. Admittedly, the juxtaposition of two

operands with no intervening operator defines multiplication in the algebraic sense. Yet, the absence of an operator symbol in a high level language is aesthetically unpleasing and could be functionally confusing.

Modifications required: a symbol used in connection with the equal sign could be used to signify assignment.

The use of a symbol (probably "*") could be introduced for multiplication in place of the blank now used. This would require a new character to indicate cross-product.

I.HALS.Defaults, Conditional Compilation and Language Restrictions

HAL/S has relatively few implementations and all are related to Shuttle software. Therefore, the distinction between basic language features and specific implementation features is not always obvious.

All of the requirements described in this section are fulfilled by HAL/S, but not all are described in the language specification. They are described in implementation oriented documents called User's Guides.

The concept of making all restrictions and limitations which are not machine-dependent a part of the language definition is basically fulfilled in HAL/S. If one considers the language definition to consist of a language specification along with an implementation-unique user's manual, then all is covered adequately.

I1.HALS.No Defaults in Program Logic
T (Satisfies)

HAL/S completely fulfills this requirement. There are no implementation-dependent defaults which affect the program logic.

I2.HALS.Object Representation Specifications Optional
T (Satisfies)

HAL/S has a large set of default capabilities for such things as data representation and procedure definition. For example, the precision of all integer and floating-point values defaults to single (Section 4.7, page 4-20; also see comments in J4). All procedures and functions default to non-reentrant (Section 3.7.2, page 3-16; Section

3.7.3, page 3-18). All default capabilities can be overridden by specific programmer action.

I3.HALS.Compile Time Variables
Fails to Satisfy

HAL/S has no compile time options which are a part of the basic language definition.

Modifications required: allow the user to specify compile time variables, especially the object machine, as a part of the basic language.

I4.HALS.Conditional Compilation
Partially Satisfies

HAL/S has a type of conditional compilation in the form of the "%macro" capability. This allows the referencing of implementation-dependent intentions to the language. Conceivably these macros could be changed to accommodate various object environments.

I5.HALS.Simple Base Language
Partially Satisfies

HAL/S has such a base of primitive features in which the remaining set of features could be defined (Appendix G) but it is not clearly identifiable.

I6.HALS.Translator Restrictions
Fails to Satisfy

HAL/S has many implementation-dependent language capabilities which are a part of the language translator itself. These are specified in an implementation unique document called a User's Guide.

I7.HALS.Object Machine ~~Restrictions~~
T (Satisfies)

Hardware limitations are not considered in the language translator or specification. For example, there is no built-in language feature to warn that the resources of the intended object machine are exceeded.

J. HALS. Efficient Object Representation and Machine Dependencies

As was stated in the introductory comments to Requirement 1, the requirements of this group of requirements are adequately fulfilled by HAL/S, but all are not described in the language specification. All capabilities and restrictions which are unique to the translator are described in an implementation oriented User's Guide.

J1.HALS.Efficient Object Code
T (Satisfies)

HAL/S does not impose run-time costs for unneeded code. If only a basic capability is required, no unnecessary checks or "hooks" (which might be needed for more intricate applications) are provided.

J2.HALS.Optimizations Do Not Change Program Effect
T (Satisfies)

The HAL/S translators provide optimized code with no loss of reliability or correctness. (See comments on RIGID attribute under J4.)

J3.HALS.Machine Language Insertions
Partially Satisfies

Interfaces with special purpose devices are accomplished through use of the %macro (Sections 11.2.1, 11.2.2, 11.2.3). This capability could, for example, allow a supervisor call, with or without an I/O list, to a specific I/O device. The code used in all such interfaces is predetermined and is extracted from a library. HAL/S does not provide a general-purpose machine language insertion capability.

J4.HALS.Object Representation Specifications
T (Satisfies)

The HAL/S user has the capability to specify that structure data be DENSE or ALIGNED. The DENSE attribute causes all bit strings to be packed into logical machine words. This packing implies all the necessary shifting and masking required to reference the data. The ALIGNED attribute starts every data entity on a word boundary, increasing the size of the structure, but allowing more efficient code to reference it. The ALIGNED attribute is the default (Section 4.5, page 4-15).

Unless prohibited, the compiler will reorder data elements of a compool or a structure into groups of like type in order to optimize storage. The user can override this capability by specifying the RIGID attribute (Section 4.3, page 4-10, rule 3; Section 4.5, page 4-15).

J5.HALS.Open and Closed Routine Calls
Partially Satisfies

HAL/S provides an "in-line" function capability (Section 11.2.1) which eliminates parameter passing; however, all standard functions are "closed". The specification of function type is made by the programmer, and is not left up to the choice of the translator (see diagram, page 11-2). The "in-line" capability is applicable only to functions and not procedures.

K.HALS.Program Environment

The variety of run time and compile time support packages associated with HAL/S contribute to making it powerful as a high level programming system. HAL/S is designed to be used for implementation of visible and reliable, yet complex, embedded computer systems.

K1.HALS.Operating System Not Required
T (Satisfies)

HAL/S has a stand-alone run capability. The primary operation mode is with an operating system which is directly related to the object machine.

K2.HALS.Program Assembly
T (Satisfies)

HAL/S modules are separately defined and separately compilable. Programs and compools are always compiled alone. Functions and procedures may be internal to a program (or function, or procedure) or they may be compiled separately (external feature) (Section 3.1).

The problems of type checking, efficient object representation and constant expression evaluation are shown by use of the "template" approach to building complex programs. If a HAL/S compilation contains invocations of external programs, procedures, or functions or references to variables in compools, the block template for those external blocks must appear in the compilation. Depending on the implementation, templates are generated automatically by the compiler for each

compilation unit as it is being compiled (Section 3.6). These templates are placed in a library where they may be combined with other compilation units to form complex programs (see comments under F4 and F6).

K3.HALS.Software Development Tools
T (Satisfies)

Associated with HAL/S is a powerful set of programming aids which are not a part of the language or the compiler. Below are listed a few of these support packages:

1. Simulation Data Files - The simulation data file for each compilation unit is a table of symbolic data used as a source of post-compilation statistics and as a run-time debugging aid.

2. HALLINK - The HALLINK program is used in System/360 applications to replace the standard link edit step. It determines the space requirements and creates the stack for each program or task. In performing this function, HALLINK makes use of the standard OS/360 linkage editor.

3. HALSTAT - The HALSTAT package is executed following the link/build step and prior to actual execution. It describes interrelationships between elements of a program complex and provides static verification.

K4.HALS.Translator Options
T (Satisfies)

HAL/S has a variety of compile time options which aid the user in debugging, testing, and documenting his program. The options include all of those mentioned, plus others. Not all are user controlled (e.g., indentation of lines to denote nesting and to aid documentation is the standard output mode).

K5.HALS.Assertions and Other Optional Specifications
Partially Satisfies

The optional features of assertions, debugging statements, definitions of units of measure, etc. are not a part of HAL/S. However, there exists in HAL/S a mechanism to insert statements of this type and have them treated as comments until such capabilities are developed. An

assertion statement could be denoted by a special letter (say "A") in column one of the input line and alternatively treated as a comment or a post-compile input statement, depending on a compile-time option.

L.HALS.Translators

The requirements of this section are for the most part implementation-dependent and therefore somewhat difficult to assess with respect to the HAL/S language.

The requirement that a translator must be written in the subject language seems somewhat arbitrary. HAL/S is sufficiently flexible that such a translator could be built but thus far none has been. However, the tone of this requirement is that it "must have been done" rather than "the capability should exist to do so".

L1.HALS.No Superset Implementations
T (Satisfies)

HAL/S has no source language features which are not defined in the language specification.

L2.HALS.No Subset Implementations
T (Satisfies)

The HAL/S implementations employ the total set of language features; there are no subsets of the language.

L3.HALS.Low-Cost Translation
Unknown

Comparison data needed to determine the relative efficiency of the HAL/S compilers is not available. There is nothing inherent in the language design that precludes efficient compilers.

L4.HALS.Many Object Machines
Partially Satisfies

HAL/S is implemented for several object machines. Nothing would preclude its being adapted to others.

L5.HALS.Self-Hosting Not Required
T (Satisfies)

HAL/S compilers are not necessarily written to execute on the object machine. The shuttle implementation employs two compilers, both run on the IBM-360. One produces 360 object code, the other AP-101 (shuttle on-board computer) code.

L6.HALS.Translator Checking Required
T (Satisfies)

HAL/S compilers perform complete syntax and type checking and do no error correction.

L7.HALS.Diagnostic Messages
Fails to Satisfy

HAL/S has a full set of compile time diagnostic messages, complete with severity levels. However, these messages are not given as a part of the language definition.

Modifications required: it would be simple to choose a basic set of diagnostic messages and make them a part of the language definition.

L8.HALS.Translator Internal Structure
T (Satisfies)

Compiler capabilities in HAL/S have thus far been determined by needs and restrictions of the various implementations.

L9.HALS.Self-Implementable Language
Fails to Satisfy

HAL/S has relatively few implementations and none have compilers written in the HAL/S language. However, there is nothing in the HAL/S language to prevent self-implementation.

Modifications required: write a self-compiling HAL/S translator.

M.HALS.Language Definition, Standards, and Control

The HAL/S language is well defined. It can be learned and understood from the available documentation - Language Specification, User's Manual, and Programmer's Guide. Current implementations of HAL/S do not employ the language control features required in this section.

M1.HALS.Existing Language Features Only
T (Satisfies)

HAL/S is a modern programming language employing features considered to be within the state-of-the-art. All modifications needed to bring HAL/S up to the TINMAN requirements are feasible using known techniques and language design methods.

M2.HALS.Unambiguous Definition
T (Satisfies)

The language specification for HAL/S is concisely written and easily understood. The syntax diagrams for each feature are accompanied by sets of general and specific semantic rules. Examples of use are provided in the more extensive Programmer's Guide. The language is described in BNF notation.

M3.HALS.Language Documentation Required
T (Satisfies)

The reference documents for HAL/S are complete and easily understood. (See comments under M2.) (Reference Appendix G)

M4.HALS.Control Agent Required
T (Satisfies)

HAL/S is under control of NASA-JSC for development and maintenance.

M5. HALS.Support Agent Required
T (Satisfies)

HAL/S is under control of NASA-JSC for development and maintenance.

M6.HALS.Library Standards and Support Required
T (Satisfies)

HAL/S is under control of NASA-JSC for development and maintenance.

#.HALS.Features Not Needed To Support TINMAN

There are many features built into the HAL/S language which are not required in TINMAN. Some of these features should be removed in order to comply with a specific requirement, and some could be removed without destroying the versatility of the language. However, some of the features not required would badly damage the language capabilities if removed.

Below are listed the features which should be removed in order to be compatible with requirements.

1. Implicit conversion in assignment statements (7.3)

2. Use of the "[cent]" sign for introduction of the "escape mechanism" in character literals (2.3.3)

Below are listed the features which could be removed without detriment to the language.

1. Multiline input format (2.4)

2. Use of "C" in column one of input line to denote a comment (2.5)

The features listed below are not specifically required but in many instances help HAL/S to fulfill the intent of a requirement. All of these features greatly add to the power of the language and if removed would severely limit its versatility.

1. REPLACE macro (4.2)

2. Matrix and vector-data types and built-in operations (4.7, 6.1.1)

3. SUBBIT operation (6.5.4)

4. %macro (11.2.2)

5. TEMPORARY variables (for other than loop control) (11.3)

6. EQUATE facility (11.5)

7. Use of radix in explicit conversion (6.5.2, 6.5.3)

8. SEND ERROR (9.2)

9. Full typing on pointer variables (11.4.1)

10. CAT operator for bit strings (6.1.2).

Section Vd - HAL/S Summary and Recommendations

//.HALS.Discussion and Summary

Considering the total set of TINMAN requirements, HAL/S does not meet DoD specifications for a common language. However, the language could be expanded to provide some of the capabilities required and thus make HAL/S a base for the design of a common language. The following table is a breakdown of HAL/S compliance with the requirements:

| REQUIREMENT | | HAL/S |
|---|---|---|
| A. | Data and Types | |
| | A1. Typed Language | T |
| | A2. Data Types | P |
| | A3. Precision | T |
| | A4. Fixed-Point Numbers | F |
| | A5. Character Data | F |
| | A6. Arrays | T |
| | A7. Records | T |
| B. | Operations | |
| | B1. Assignment and Reference | T |
| | B2. Equivalence | T |
| | B3. Relationals | T |
| | B4. Arithmetic Operations | P |
| | B5. Truncation and Rounding | P |
| | B6. Boolean Operations | T |
| | B7. Scalar Operations - On arrays | T |
| | B8. Type Conversion - Implicit | F |
| | B9. Type Conversion - Explicit | P |
| | B10. I/O Operations | P |
| | B11. Power Set Operations | F |
| C. | Expressions and Parameters | |
| | C1. Side Effects | T |
| | C2. Operand Structure | T |
| | C3. Expressions Permitted | T |
| | C4. Constant Expressions | T |
| | C5. Consistent Parameter Rules | T |
| | C6. Type Agreement in Parameters | T |
| | C7. Formal Parameter Kinds | T |
| | C8. Formal Parameter Specification | F |
| | C9. Variable Numbers of Parameters | F |
| D. | Variables, Literals and Constant | |
| | D1. Constant Value Identifiers | T |
| | D2. Numeric Literals | U |
| | D3. Initial Values of Variables | T |

| | | |
|---|---|---|
| D4. | Numeric Range and Step Size | F |
| D5. | Variable Types | P |
| D6. | Pointer Variables | T |
| E. | Definition Facilities | |
| E1. | User Definitions Possible | P |
| E2. | Consistent Use of Types | F |
| E3. | No Default Declarations | F |
| E4. | Can Extend Existing Operators | F |
| E5. | Type Definitions | F |
| E6. | Data Defining Mechanisms | F |
| E7. | No Free Union or Subset Types | T |
| E8. | Type Initialization | F |
| F. | Scope and Libraries | |
| F1. | Separate Allocation and Access Allowed | T |
| F2. | Limiting Access Scope | T |
| F3. | Compile Time Scope Determination | T |
| F4. | Libraries Available | T |
| F5. | Library Contents | T |
| F6. | Libraries and Compools Indistinguishable | T |
| F7. | Standard Library Definitions | T |
| G. | Control Structures | |
| G1. | Kinds of Control Structures | P |
| G2. | The GOTO | T |
| G3. | Conditional Control | P |
| G4. | Iterative Control | P |
| G5. | Routines | F |
| G6. | Parallel Processing | T |
| G7. | Exception Handling | T |
| G8. | Synchronization and Real-Time | T |
| H. | Syntax and Comment Conventions | |
| H1. | General Characteristics | T |
| H2. | No Syntax Extensions | T |
| H3. | Source Character Set | T |
| H4. | Identifiers and Literals | T |
| H5. | Lexical Units and Lines | P |
| H6. | Key Words | T |
| H7. | Comment Conventions | P |
| H8. | Unmatched Parentheses | T |
| H9. | Uniform Referent Notation | T |
| H10. | Consistency of Meaning | F |
| I. | Default, Conditional Compilation and Language Restrictions | |
| I1. | No Defaults in Program Logic | T |
| I2. | Object Representation Specifications Optional | T |
| I3. | Compile Time Variables | F |
| I4. | Conditional Compilation | P |

| | | |
|---|---|---|
| I5. | Simple Base Language | P |
| I6. | Translator Restrictions | F |
| I7. | Object Machine Restrictions | T |
| J. | Efficient Object Representations and Machine Dependencies | |
| J1. | Efficient Object Code | T |
| J2. | Optimizations Do Not Change Program Effect | T |
| J3. | Machine Language Insertions | P |
| J4. | Object Representation Specification | T |
| J5. | Open and Closed Routine Calls | P |
| K. | Program Environment | |
| K1. | Operating System Not Required | T |
| K2. | Program Assembly | T |
| K3. | Software Development Tools | T |
| K4. | Translator Options | T |
| K5. | Assertions and Other Optional Specifications | P |
| L. | Translators | |
| L1. | No Superset Implementations | T |
| L2. | No Subset Implementations | T |
| L3. | Low-Cost Translation | U |
| L4. | Many Object Machines | P |
| L5. | Self-Hosting Not Required | T |
| L6. | Translator Checking Required | T |
| L7. | Diagnostic Messages | F |
| L8. | Translator Internal Structure | T |
| L9. | Self-Implementable Language | F |
| M. | Language Definition, Standards and Control | |
| M1. | Existing Language Features Only | T |
| M2. | Unambiguous Definition | T |
| M3. | Language Documentation Required | T |
| M4. | Control Agent Required | T |
| M5. | Support Agent Required | T |
| M6. | Library Standards and Support Required | T |
| Totals | T (Fully Satisfies) | 59 |
| | P (Partially Satisfies) | 18 |
| | F (Fails to Satisfy) | 19 |
| | U (Unknown) | 2 |

Many of the features required in the TINMAN have not been found necessary for embedded spacecraft computer systems. These include extensibility, generic procedures, and recursion. If these requirements were removed, HAL/S would only fail in the following ten areas:

1. Fixed-point numbers (A4, D4)

2. Character sets as enumeration types (A5)
3. Implicit conversions (B8)
4. Power set operations (B11)
5. Default declarations (E3)
6. Consistency of meaning (H10)
7. Compile time variables (I3)
8. Translator restrictions (I6)
9. Diagnostic messages (L7)
10. Self-implementable language (L9).

As possible modifications to the language, none of these are insurmountable problems.

HAL/S, considered as a base language for the DoD HOL:

a. Is compliant with the majority of existing TINMAN requirements

b. Could be easily changed to meet approximately 13 of the requirements it does not fully meet

c. Could be changed with moderate difficulty in approximately 12 of the requirements it does not fully meet

d. Would be very difficult, but not impossible to change in approximately 14 of the remaining requirements it does not fully meet

e. Is a viable base technically, but is probably a poorer choice than some of the larger, more widely used languages.

## Section VI - COMMENTS ON TINMAN REQUIREMENTS

### Section VIa - TINMAN General Comments

1. Many of the TINMAN requirements are arbitrary and matters of personal taste. To a large extent, this is unavoidable since good language design and definition is an art and even the experts do not agree on what "good" is.

   The following are concepts on which there is legitimate disagrement:

   a. Defaults

   b. Implicit conversions from one data type to another

   c. Designation by the user of open or closed subroutines.

2. There are various contradictions in the TINMAN requirements. Some examples are:

   a. Within a specific requirement, consider B2 and its comments on comparison "(regardless of type)" in the header and "restricted to data of the same type" in the commentary.

   b. Across requirements, (G4) wants a special structure, namely, WHILE DO; another requirement (H1) says that there should be no special cases.

   c. Across requirements, (I5) specifies a simple base language, but one which satisfies all other requirements. The 98 requirements for TINMAN cannot be fulfilled by a simple base language.

3. Much of the text of the requirement is a justification rather than a clarification. As examples in (A3) the sentence beginning "Machine Independence", in (A4) the first sentence, and in (B11) the first sentence.

4. Some languages will satisfy - or fail to satisfy - some

requirements by cascading or by default, i.e., the requirements are not independent. This could make the counting of Fully, Partially and Fails for each language somewhat misleading.

As examples,

a. E2, 4, 5, 6, 8 are failed by COBOL by default.

b. H2 and H9 are satisfied by COBOL by default.

c. Failure of a language with respct to E1 will automatically mean failure with respect to E4.

5. Some requirements are ambiguous. For example, "fixed-point" in A4 and D4, and "equivalence/identity" in B2 are ambiguous.

6. Many of the requirements pertain to the implementers and not to the language specification. Certainly it is appropriate for TINMAN (and its successor lists of requirements) to impose requirements on the implementers, but they should be separated from the language requirements. Furthermore, many of these implementation requirements are vague and/or not measurable. For example; wanting good object code is obviously desirable, but there is no concrete way within the scope of this study to evaluate a language with respect to that requirement. As another example; it is certainly desirable that numeric data should be converted the same way at compile time and object time, but that has nothing to do with language specifications.

7. An almost fatal effect in the TINMAN requirements is the failure to provide a ranking of requirements, in terms of their importance to the intended area of application. To the extent that tradeoffs are required in the language design, it should be made clear which requirements are more important. In doing this, the translator and/or maintenance requirements should be separated and ranked separately.

8. There is a significant emphasis on libraries. While this is desirable, it seems to show a lack of faith in the viability, effectiveness, and usefulness of the eventual language. Obviously, no language is self-sufficient enough to make libraries unnecessary; however, if the language is really going to be good, TINMAN appears to be over-emphasizing this need.

Section VIb - TINMAN Specific Comments

A.TINMAN.Data and Types

The fixed-point requirement seems to force an almost redundant capability in implementations which also have integer and real facilities. It would seem that the HAL/S approach of making them mutually exclusive based on hardware requirements would be an acceptable compromise.

A4.TINMAN.Fixed-Point Numbers

This requirement is ambiguous. In general terminology, fixed-point numbers are sometimes interpreted as numbers between plus and minus 1; in other cases, fixed-point numbers are of the form NNN.NNN with any number of digits on either side of the decimal point.

B1.TINMAN.Assignment and Reference

The requirements say "The user will be able to declare variables for all data types". This seems to be a comment which has nothing to do with the basic requirement of B1 and which is already dealt with under requirement A2.

B9.TINMAN.Changes in Numeric Representation

The requirement that: "There will be a run time exception condition when any integer or fixed-point value is truncated" is ambiguous. It is not clear whether the requirement means that there should be an automatic transfer to some predetermined routine whenever this error occurs, or whether the programmer can specify what is to happen if this error occurs. COBOL, for example, provides the "ON SIZE ERROR" clause in the five arithmetic verbs which permits the programmer to determine what is to happen if truncation or other errors (such as division by zero) occur.

C.TINMAN.Expressions and Parameters

The "generic procedure" capability of TINMAN seems to violate the purpose of having a strongly typed language with multiple compiler checks for type mismatches (A1, C6). It would cause the compiler to be cumbersome and would be very costly to the user at compile time and run time.

D1.TINMAN.Constant Value Identifiers

The use of identifiers to represent constant values is desirable and present in most HOLs. However, the implication that section and paragraph names have constants assigned to them does not make sense within the overall spirit of TINMAN's avoidance of tricky coding.

D2.TINMAN.Numeric Literals

The requirement that "numeric constants will have the same value...in both programs and data..." is a directive to the implementer and not a language specification.

E.TINMAN.Definition Facilities

The TINMAN requirement for user definitions of data and operations seems to contradict the basic desire for readability and maintainability. The capability to extend a language might be useful but it would have to be very stringently controlled. This would also force very large, general compilers which could be beyond the hardware capabilities for some implementations.

G.TINMAN.Control Structures

The recursive capability (G5) seems to be another of those requirements which yield cumbersome compilers and add to execution time. HAL/S does not provide recursiveness because of these penalties and the knowledge that on-board aerospace applications have little need for it. In time-critical on-board aerospace applications (the indefinite space requirements needed for recursion) cannot be afforded, nor can the possibility of an error condition due to overflow of stack sizes in real-time be allowed.

I5.TINMAN.Simple Base Language

It should be obvious that this requirement seems incompatible with the others; to satisfy all the other TINMAN requirements will require a powerful - and not a simple - language.

J.TINMAN.Efficient Object Representation and Machine Dependencies

The general tone of the requirements of I and J are more translator

oriented than general language definition oriented. This requires looking at implementations and considering their needs and restrictions as opposed to looking at the general capabilities of a language.

The requirement to "not impose run time costs for unused generality" seems to be in direct conflict with the generic capabilities of C8, C9, and G5.

J3.TINMAN.Machine Language Insertions

There would be little difficulty in modifying most languages to satisfy the part of this requirement that involves inclusion only in compile time conditional statements if they existed (per requirement I4). However, this part of requirement J3 seems to defeat the purpose of allowing machine language instructions in the source program at all.

J5.TINMAN.Open and Closed Routine Calls

This appears to be an extremely undesirable requirement. While it is useful as long as a program is running on a specific machine with a known translator, its use can hamper efficient portability. The programmer might specify an open subroutine for a particular configuration and translator; yet when that program is moved to another machine and/or translator, it might be far more efficient to have a closed subroutine. Since the programmer has already specified which it is to be, the translator is unable to make the determination. It would be far better to leave this decision entirely in the hands of the translator.

L3.TINMAN.Low-Cost Translation

As an implementation requirement, this is in at least partial conflict with L4.

# Some Characteristics for a common DOD Programming Language and their relation to PL/I

Dr. B. L. Marks
MP 185
IBM United Kingdom Laboratories Limited
Hursley Park Winchester

Abstract

Many of the needed characteristics of 'TINMAN' are inter-related. This paper discusses strong typing, pointer machanisms, user defined types, storage management, parallel processing and real time in terms of both function and potential PL/I extensions.

The needed characteristics for 'TINMAN' are related, for example storage management is very different in the presence of parallel processing than in its absence. This paper traces a subset of those relations. The examples are given in terms of subsetting and extending ANS Standard PL/I. This has the advantage of using a familiar syntax and relating 'TINMAN' to historical solutions. The potential extensions are illustrative and there is no claim that they are well defined. One person in a day can invent enough language to keep definers, implementors and standards committees busy for years.

PL/I is a typed language, in the sense that the type of data accessed by any reference in a correct program can be determined at compile time (A language that does not meet this requirement is scarcely compilable.) Difficulties arise with erroneous programs, that use pointers. On normal hardware the check that the programmer's assertions about the objects referenced match the actual objects referenced dynamically, is comparatively expensive. The only implementation of PL/I incorporating this check in full, executes several times more slowly than the same PL/I can be executed after compilation to code that does not check. (Ref 1)

If the check is not made the programmer may find it hard to detect his error, or may even make his program dependent on the error, because the semantics of erroneous execution are often dependable. The simplest solution to this difficulty is to subset PL/I to a level where the error cannot be written without detection at compile time. The subsetting rule would be that a particular POINTER (or OFFSET) variable must only point to a single data type. The compiler could detect this single data type by inspecting the pointer usage. It would detect the error of assignment between pointer variables that address different data types. We should also assume there is a mechanism for making checks similar to those the compiler makes, but across separate compilations. Hence we can note that subsetting alone is sufficient to make PL/I a strongly typed language.

It is noteworthy that many of the 'TINMAN' characteristics aimed at legibility and compilability can be obtained by subsetting PL/I, for example the absence of abbreviations, the absence of implicit conversion, and declaration before use.

In the case of typing it will help if the subsetting is accompanied by a small extension. This would permit the declaration of a pointer to describe what it was to point at.

```
DECLARE P POINTER TO (FLOAT);
```

This would be redundant input, except as a check, but would be an aid to legibility.

Of course the function that is subset has some use, or it would not be in the standard PL/I, and it may be necessary to compensate for it. The usual use for a pointer that addresses different types is the handling of 'self-defining records'. In practice it often happens that one has a pointer to an object without knowing the object type, for instance after reading the next record from a data set into a buffer. Usually one will know that whatever alternative is in the buffer, it will start with a byte, say, which has a value identifying the data type. Call this a tag. It is not incorrect to use the pointer first to address the tag and then the whole record.

```
DECLARE TAG CHAR (1) BASED (Q)
DECLARE 1 WHOLE BASED (Q),
     2 TAGA CHAR (1),   /* CASE A */
     2 XXX FLOAT;       /* say */
IF Q->TAG ='A' THEN Y = Q -> XXX;   /* is not incorrect */
```

The discrimination could be arbitrarily complex, depending for example on several common fields in the left fields of the alternative records. The extent to which similar structures have a commonly addressable part is known as the left-to-right equivalence problem. (Ref 2)

The coding above would not be acceptable to a PL/I subset that achieved strong typing solely by subsetting, since Q points to different types. To provide both strong typing and the ability to handle self-defining records one might add to PL/I the concept of a union of data types. The attribute ONEOF would indicate that the components of a structure were alternatives rather than members of an aggregate.

```
DECLARE 1 AAA,
     2 BBB CHAR (1),  /* The tag */
     2 CCC ONEOF,
          3 XXX FLOAT WHEN (BBB='A')
          3 YYY FIXED WHEN (BBB='B')
```

The clauses that say which alternative corresponds to each tag value are necessary so that they can be enforced when the objects are created and used. This has re-introduced the execution-time check that we set out to avoid, but in a more controlled form. A naive implementation would make a check on every reference to variables in the union, but this can be improved on. If the compiler has mechanisms for recognizing common sub-expressions (reference 3) then these will also recognize tests that are unnecessary because of other tests earlier in the execution. Alternatively, or also, one can introduce syntactic constructs such as the TAGCASE construct of CLU (reference 4) which syntactically force sections of the program to contain only references with known tag values. Current IBM implementations of PL/I provide a form of case

selection (reference 5). This construct could be extended to handle type discrimination. By such approaches the strong typing of unions can be made efficient. Notice that with this sort of union it is not meaningful to assign to the tag, because the tag is slaved to the data type and the data type is unchanged during the lifetime of the data. This suggests a more elegant extension where the tag is known only to the language system.

```
DCL 1 CCC ONEOF,
      2 XXX FLOAT,
      2 YYY FIXED;
```

An alternative syntax could be modelled on the GENERIC attribute, which has WHEN clauses. Whether one thinks this is preferable probably depends on ones views about using level numbering to describe structures in general.

This cannot subsume the inelegant form if one wants to read existing records where the interpretation of the tag is known only to the programmer.

One can envisage uses for a union where the data type of the content of the cell did change during the lifetime of the cell. Consider writing the part of a compiler that builds a dictionary item relating to each variable in the program being compiled. The sort of information that is held in an item will vary during compilation. For example the symbolic name may be needed only in the early stages and the compiler assigned internal number only needed in the later stages. Storage will be saved if the same cell can hold first the symbol and then the number. This example cannot be validly programmed on PL/I although the user with a permissive implementation and a knowledge of the physical data mappings might wangle it.

The obvious extension is another union type. The difference from the ONEOF union is that the storage allocated when the cell is created has to be large enough for the largest of the alternatives whereas the storage for a ONEOF is determened by the particular alternative allocated. Since existing records can be accessed with the ONEOF type, the tag for a UNION can be implicit.

```
Example:  DCL 1 DDD UNION,
                2 EEE CHAR (4),
                2 FFF FIXED;
```

The tag contains the information about which data type alternative currently occupies the cell. Assignment to the cell may alter the current type. There is no requirement for implicit conversion, so that it is an error to attempt a fetch of the wrong type.

```
Example:  DDD.EEE ='9999'; /* Current type becomes type of EEE */
                J = DDD.FFF;   /* Error, as next successive instruction*/
```

An interesting case occurs when the information in the tag about currency is the only information that a particular program needs. The traditional example is:

```
DCL 1 COLOUR UNION,
      2 RED,
      2 BLUE,
      2 YELLOW;
```

The only storage required is for the tag. A suitable syntax for setting and testing the tag would be COLOUR = BLUE; IF COLOUR =BLUE THEN etc.

To fully exploit strong type checking it would also be necessary to extend PL/I to provide user definition of new data types. Without new data types the user who decided on using FLOAT for a variable representing a length, and FLOAT also for a variable representing a temperature, would not be able to use the type checking to detect accidental attempts to multiply lengths by temperatures. There are also other very good reasons for providing user definition of types.

The constituents of such an extension are probably not too contentious:

1. There is a way to introduce the symbolic name of the new type.

2. There is a block within which the new type is defined.

3. The data structure of the new type is defined in terms or system supplied types and user types. This structure is visible inside the block but not visible from outside.

4. Operations applicable to the new type are defined. (Possibly one should distinguish specifically the case where there are no new operations but existing operations on the aggregate that forms the type are allowed.)

5. Some of these operations must be recognisable by the compiler, so that they can be invoked impicitly. For example, to support assignment between two instances of the data type.

6. Some of the operations are visible from outside the static scope and some may not be.

One natural map of these constituents to PL/I syntax is a follows:

a. A procedure is used to introduce the name and scope.

```
FLIGHT:PROCEDURE;
       /* Definition in here */
END;
```

b. The data structure of the new type is declared.

```
FLIGHT:PROCEDURE;
   DCL 1 FLIGHT,
         2 FARE FIXED,  /* FIXED is a system type */
         2 ORIGIN AIRPORT,  /* AIRPORT is not a system type */
         2 CONTINUATION POINTER TO (FLIGHT);  /* At least this much
           recursion should be allowed */
END;
```

c. Functions applicable to the type are written as internal procedures.

```
FLIGHT:PROCEDURE;
     DCL 1 FLIGHT, etc;
          SETFARE:PROCEDURE(A,B);
               DCL A FLIGHT;
               DCL B FIXED;  /* New fare in dollars, say */
               IF B < A.FARE THEN
                    CALL REDUCTION/*Constituents of a
                         FLIGHT are visible inside procedure FLIGHT */
               A.FARE = B;
          END SETFARE:
END;
```

d. The spelling of some function names makes them recognisable to the compiler. For example the compiler would compile IF A = B THEN ... to a call on the EQUAL procedure within FLIGHT if both A and B were FLIGHTS. The closest analogy to this in existing PL/I is the treatment of BUILTIN function names.

e. Classical nested scope rules would make the operations inside the data definition procedure inaccessible from outside that procedure. Extra syntax is necessary to specify where this is not the required scope. EXTERNAL provides this facility in existing PL/I, in a gross way, which suggests a new syntax EXTERNAL (block name) to indicate that the scope is to be as if the symbol was introduced immediately externally to the named block.

The syntax introduced above is not as concise as the syntax could be if it were entirely new. However using existing syntax as a basis will help the programmer carry his knowledge of the existing language over to

the extension.

It will be no advantage to add strong checking and type definition to PL/I if they cannot be used with safety. Safety is primarily a question of implementation rather than language - the highest level of PL/I implemented by IBM has been impemented with safety (Ref 1). So the question reduces to combining safety with a minimum execution time checks. A powerful language cannot expect to eliminate all execution time checks, for example range checking of general subscripts in array references, but the choice of language can reduce such checks.

The definition of safety that I will use is 'For any invalid program there is a valid program that has the same effect'. Roughly translated this means 'You cannot cause the system to crash', if one assumes correct implementation or valid programs. Under this definition programs are as well-contained as they can be, given that programmers make mistakes, but the implementation will not necessarily provide the best possible execution time messages.

There is a another definition of safety that says 'Every program that the compiler will accept has a fully defined execution'. I suggest this definition of safety is too demanding. For instance it would require every variable to be initialized to a known value unless there was an execution time check that the first use of the variable was a store to it rather than a fetch from it. This may be desirable as an option but with today's hardware probably should not be mandatory.

Any language with dynamic data structuring is a challenge to implement safely. The risk is that when a piece of storage is reused some pointer values that addressed its first incarnation may still exist, and be accidently used, during the subsequent reuse of the storage. some solutions are:

1. Never reuse storage unless compile time calculations prove it to be safe. This may pessimise the storage needed, compared with execution time allocation, even if the sizes of the items to be mapped are known at compile time. This method will not reuse storage for items of which the size is not known until execution, or items which are explicitly allocated and freed at execution time. However it does ensure efficient addressing and we can expect any compiler to use this compile time mapping as part of its armoury.

2. Do a scan-mark garbage collection to find storage that has no pointers addressing it, and hence can be reused. This is unsuitable for real time applications because the work cannot be spread thinly over time.

3. Maintain all the pointers to a given piece of storage on one ring. This maintenance is awkward and doubles the storage for pointers, but the work is spread over time.

4. Use the scope rules of the language to prevent a pointer addressing some variable that has a shorter lifetime than the pointer itself. In PL/I for instance this would mean subsetting the language so that an AUTOMATIC POINTER in a containing procedure could not be assigned the address of an AUTOMATIC variable of the contained procedure. This is a compile time check. This method does not suffice where the lifetime of a variable is determined dynamically, e.g. by ALLOCATE and FREE statements.

5. Maintain a separate pool for each user defined data type. The pool itself would have the same lifetime as the data type definition. Items within a pool can be dynamically allocated, and reused, without risk of violating type checking. The reuse of an item is equivalent to copying its address from one pointer to another.

The combination of methods 1, 4 and 5 represents a practical compromise between efficiency and ease of coding and debugging. But it still leaves a question - what happens it there is insufficient storage at execution time? One could restrict the language so that this condition could not occur. However if the real world problem does make dynamic demands on the computation, then restricting the language only moves the problem to the coder. He will have to define fixed size aggregates and sub- allocate them. The condition of storage overflow will still be possible, although it may appear as a range error on a subscript, for example. Notice also that guessing the maxima of several demands does not give efficient allocation, unless the maxima occur simultaneously in execution. So this is one case where restricting the language is not the answer.

There are some implementation difficulties with treating storage overflow analogously with arithmentic overflow, i.e. raising a condition that the executing program can handle. (Where does the storage to implement the handling come from?) However, if we assume these implementation difficulties can be overcome, this approach would allow the program to avoid disaster by some action like transferring work to a backup computer, cancelling a low priority task, or explicitly freeing storage.

The presence of multiple tasks complicates the storage problem. If the classical stack mechanism is used for procedure linkage and AUTOMATIC then there will be multiple stacks. (One stack equals one task will suffice here for a definition of task.) If the implementation allows for an individual stack to be non-contiguous then the position is not worsened much by tasking. If the stack is to be contiguous it may be

necessary to fix its maximum size when the task begins. The presence or absence of address translation hardware will also be a complicating factor.

We assume that switching from the execution of one task to executing another is not restricted to predictable places in the program; for example it may be the result of a high priority interrupt from an external source. Hence any mechanism for sharing storage across tasks is faced with the need to serialize its activities.

The discussion above has led us into real time questions from the implementation viewpoint. It will be necessary to approach from the direction of language, as well.

Three crucial questions with respect to tasks are:

1. What states can a task enter?

2. How do we address a task?

3. How do we synchronize tasks?

To answer the first question, we first introduce a 'running' state. A task in the running state is capable of progressing through the execution of instructions. No guarantees are made about the rate at which it actually does so. In the 'suspended' state the task is guaranteed not to progress. In theory these two states are probably enough. For example if we want to do something as possible after midnight the task could be started but suspended until midnight. And if we wanted a task to have no further effect we could suspend it indefinitely. However it is usual to have 'scheduled' and 'non-existent' (or 'cancelled') states as well. A scheduled state is similar to a suspended state where the task is suspended prior to the first instruction of the task. Hence in the example we would schedule the task for midnight. The advantage of scheduling is that a scheduled task consumes very little storage, perhaps just a few words in a scheduling table.

A PL/I real time extension would require new statements for inducing the changes of state. The verb for cancelling a running task should differ from the verb for preventing the execution of a scheduled task because the latter is a much safer thing to do. (The running task may be part way through a sensitive operation.)

The second question asks how a task is addressed. One solution is for the language to treat each task like a data instance, so that the normal name resolution mechanism is sufficient. If there are many similar tasks this will imply arrays or dynamic allocation of tasks. It may be

conceptually simpler to introduce the task variable. The task variable can be associated with different tasks at different times, and is used as an indirect reference to the associated task. Obviously there can only be one task associated with a given task variable at a given time. In this sense the task variable is like an engine allocated to process the task. In a PL/I extension such task variables would be the operands in the state changing statements.

The synchronization question is more debatable. We will consider three of the possibilities:

a. A simple primitive. The most widely known is the semaphore, with its P and V operations. Some existing PL/I implementations use EVENTS with posting and waiting operations. This is analogous to the semaphore but not identical. It is not part of the ANS PL/I Standard.

b. A simple compile time construction. Many synchronization problems consist of no more than establishing 'exclusive' regions in the code, where the tasks operate serially. If a bracketed construct is introduced into the language (e.g. a MONITOR) then the compiler can provide the properly balanced execution time primitives to achieve these regions.

c. The path expression (Reference 6). The path expression is a constraint on the order of execution of some operations. The constraint applies irrespective of which tasks execute the operations. For example the path expression

```
DO WHILE(TRUE);
     PRODUCE;
     CONSUME;
END;
```

would synchronize a multiple producer/consumer single buffer problem.

Path expressions allow for alternative next operations, and thus are a natural place to specify priorities or fairness rules if there are alternative sequences possible.

It is a judgement question as to which of these possibilities is preferable for a DOD language. The argument against the minimum primitive is that it may prove difficult and error-prone to solve simple problems. The argument against higher level constructions is that a construct selected now may appear a poor choice when we have a few years more experience.

None of the possibilities allow the compiler to make a perfect job of checking that the synchronization coded is sufficient to serialize the use of all the variables shared between tasks. If safety is not to be compromised the computer must provide implicit serialization where it cannot recognise any explicit serialization, and this may be expensive at execution time. The higher level constructs ameliorate this problem.

The syntax for PL/I extension depends on the possibility chosen. For semaphores, the EVENT, WAIT and POST syntax is suitable. For monitors a synchronization variable would be associated with a scope, i.e.

```
BEGIN (synchronization-variable);
     Critical region
END;
```

For path expressions the data type definition extension would be a prerequisite. The path expression, using the flow of control statements of PL/I, would be written in the body of the data defining procedure.

Since most of this paper is about extending PL/I I should finish by repeating my earlier remark about subsetting. The initial approach to matching PL/I with 'TINMAN' should take the format of subsetting strongly. The removal of defaults and implicit activities, which were aimed at ease of coding, will bring PL/I in line with the readability and discipline required by 'TINMAN'.

1. 'OS PL/I Checkout Compiler General Information', IBM, order number GC33-0003-4.

2. MacLaren, M.D. 'Data Matching, Data Alignment and Structure Mapping in PL/I' Sigplan Notices on Programming Languages Vol 5 No 12 Dec 1970

3. Allen, F.E. 'Bibliography on Program Optimization', IBM Thomas J Watson Research Center, Yorktown Heights, New York.

4. Liskov, B. 'An Introduction to CLU', Computation Structures memo no. 33, Laboratory for Computer Science, MIT, Cambridge, Mass, 1976

5. 'OS PL/I Optimising Compiler General Information', IBM, order numberGC33-0001-3.

6. Habermann, A.N. 'Path Expression' Carnegie-Mellon University, Pittsburgh, PA, 1975.

PL/I and "Tinman" 21 October 1976

We have considered to what extent the ANS standard definition of PL/I meets the "Tinman" requirements, and where it does not, the modifications needed to make it do so. Some of these modifications can be met by a subset of the ANS language: other require extensions. Compliance to the 97 points of "Tinman" can be summarised as follows:

| | complete | partial | none |
|---|---|---|---|
| ANS Standard PL/I: | 50 | 35 | 12 |
| subset: | 65 | 20 | 12 |
| extensions: | 94 | 1 | 2 |

The three points which the modified language does not meet are:

| | | |
|---|---|---|
| C8 | "generic" procedures | ) we do not believe |
| C9 | variable number of parameters | ) these are required. |
| B7 | array operations | we think it best to not include any built-in array operations, except assignment and indexing. |

We have not yet prepared a detailed list of PL/I features not required to meet the "Tinman" requirements. But we are confident that this list contains a substantial part of the complexity of PL/I and that the reduced language will meet the letter and spirit of the requirements L2 and L3.

RF Maddock
IBM UK Laboratories
Hursley Park
Winchester, England

BL Marks
IBM UK Laboratories
Hursley Park
Winchester, England

TC Spillman
IBM FSD
OWEGO, NY

Compliance summary

C = complete, pP = partial, N = none, + = better but same category

| | full ANS PL/I | Subset | Extensions | Comments |
|---|---|---|---|---|
| A1 | P | C | | |
| A2 | C | | | |
| A3 | C | | + | |
| A4 | P | | C | |
| A5 | N | | C | |
| A6 | P | C | | |
| A7 | P | | C | |
| B1 | C | | | |
| B2 | C | | | |
| B3 | C | | | |
| B4 | C | | | |
| B5 | P | C | | |
| B6 | C | | | |
| B7 | C | P | | remove array ops |
| B8 | P | C | | |
| B9 | C | | | |
| B10 | P | | C | |
| B11 | P | | C | |
| C1 | C | | | |
| C2 | P | C | | |
| C3 | C | | | |
| C4 | C | | | |
| C5 | C | | | |
| C6 | P | C | | |
| C7 | P | | C | constant parms |
| C8 | P | N | +? | "generic" |
| C9 | N | | | variable number of arguments |
| D1 | P | | C | |
| D2 | C | | | |
| D3 | C | | + | |
| D4 | P | | C | |
| D5 | P | | C | |
| D6 | P | + | C | |
| E1 | N | | C | |
| E2 | N | | C | |
| E3 | P | C | | |
| E4 | N | | C | |
| E5 | N | | C | |
| E6 | N | | C | |
| E7 | C | | | |
| E8 | N | | C | |

| | | | | |
|---|---|---|---|---|
| F1 | C | | | |
| F2 | P | | C? | |
| F3 | C | | | |
| F4 | C | | + | |
| F5 | C | | + | |
| F6 | C | | | |
| F7 | C | | | |
| | | | | |
| G1 | P | | C | |
| G2 | P | C | | |
| G3 | P | | C | |
| G4 | P | | C | |
| G5 | C | | | |
| G6 | N | | C | |
| G7 | C | + | | |
| G8 | N | | C | |
| | | | | |
| H1 | P | C? | | |
| H2 | C | | | |
| H3 | C | | + | |
| H4 | P | | C | break numbers? |
| H5 | P | C | | |
| H6 | C | | | reserve <u>some</u> kwds |
| H7 | C | | | |
| H8 | N | C | | |
| H9 | C | | | |
| H10 | P | + | C | := makes complete |
| | | | | |
| I1 | P | | C | |
| I2 | C | | + | |
| I3 | C | | | |
| I4 | C | | | |
| I5 | P | C | | |
| I6 | C | | + | |
| | | | | |
| J1 | P | C | | |
| J2 | C | | | |
| J3 | P | | C | |
| J4 | N | | C | |
| J5 | P | | C | |
| | | | | |
| K1 | P | C | | |
| K2 | C | | | |
| K3 | C | | | |
| K4 | C | | | |
| K5 | C | | + | |
| | | | | |
| L1 | P | | C | |
| L2 | P | C | | |
| L3 | P | C | | |
| L4 | C | | | |
| L5 | C | | | |
| L6 | C | | | |
| L7 | C | | + | |
| L8 | C | | | |
| L9 | C | | | |

```
M1      C
M2      C C
M3      C C
M4      C C
M5      C C
M6      C
```

Compliance details for ANS Standard PL/I

General remark : references to PL/I mean the ANS Standard PL/I.

A1. Typed Language

Degree of Compiliance – Partial.

Explanation – the type of variables may be explicitly declared or acquired by default.

Scope of Modifications – Disallow acquiring type by default.

A2. Data types – integer, real, Boolean etc.

Degree of Compliance – Complete.

Explanation – Integer is specified as special case of fixed point. Boolean is special case of bit strings.

Scope of Modifications – none.

A3. Global precision specification

Degree of Compliance – complete.

Explanation – precision of variables can be specified in the manner described. Precision of operations is determined by the precision of operands rather than by a separate declaration. The intent of the requirement is considered to be met.

Scope of Modifications – none.

A4. Fixed Point.

Degree of Compliance – Partial.

Explanation – Range and stepsize are determined by declared precision and scale, and are thus integer powers of the number base.

Scope of modifications – consider adding a range specifications different from precision. If a stepsize other than a power of the base is required then consider adding language to specify it. (Note that this would imply multiplications rather than shifts for the general case of mixed stepsizes.)

A5. Character sets as enumeration types.

Degree of Compliance - Fails.

Explanation - A single character set is defined by each language implementation. Built-in functions are provided to translate from one character set to another.

Scope of modifications - Modifications required to allow data types defined by enumeration are described in F6.

A6.

Degree of Compliance - Partial.

Explanation - lower subscript bound as well as upper subscript bound determined at entry to array allocation scope if lower bound is not compile time constant.

Scope of modications - remove capability to allow variable expressions as subscript lower bounds.

A7.

Degree of Compliance - Partial.

Explanation - if the alternative structures differ only in terms of extents, e.g. array dimension, then PL/I complies. The more general alternative structures are provided using the left to right equivalence of different structures, but the safety requirement is not met.

Scope of modifications - add ability to specify alternative structures in one declaration, together with the expression that discriminates among them. In conjunction with constrained pointers (D6), this provides the specified safety. See 'Some Characteristics for a Common DOD Programming Language and their Relation to PL/I' by Dr BL Marks for examples.

B1.

Degree of Compliance - Complete.

Explanation - the existing data types can be assigned and referenced.

Scope of modifications - modifications required for encapsulated type definitions are discussed in B5.

B2.

Degree of Compliance - Complete.

Explanation - Comparison of atomic data is provided through the equality operator. Addresses can be compared to determine identify of objects.

Scope of modifications - none.

B3.

Degree of Compliance - Complete.

Explanation - Relational operations are defined for all numeric data types.

Scope of modifications - Defining types by enumeration is discussed in B6.

B4.

Degree of Compliance - Complete.

Explanation - PL/I provides operators + - * / ** and built-in functions to more precisely control scale and precision of arithmetic operations.

Scope of modifications - the operations provided by PL/I may be more than required, and a suitable sunset could be selected. For example, exponentiation could be restricted to an integer power.

B5.

Degree of Compliance - Partial.

Explanation - Programs which truncate most significant digits will raise an on-condition indicating the program is in error. Implicit truncation of least signficant digits can occur for integer and fixed point. Built-in functions can be used to make these operations explicit.

Scope of modifications - Introducing restrictions on arithmetic operations performed on data items with different precisions or scales will eliminate implicit trunction.

B6.

Degree of Compliance – Complete.

Explanation – PL/I provides operators for "and", "or" and "not" and a built-in function for "or". An implementation is permitted to evaluate expressions using short circuit mode as described.

Scope of modifications – none.

B7.

Degree of Compliance – Complete.

Explanation – Conformability requires identical dimensions except that items of lower rank will be promoted to match one of a higher rank if this makes them conform. As a special case, a scalar is conformable to any array.

Scope of modifications – PL/I does meet this requirement. But this function is not a desirable one – it is better for the programmer to define his aggregate operations explicitly. Eliminate all aggregate operations except assignment of identical aggregates and of scalar to array.

B8.

Degree of Compliance – Partial.

Explanation – Explicit conversion operations are provided but implicit type conversion is also included.

Scope of modifications – eliminate implicit type conversions, but consider retaining this feature for literal constants and across assignment operations.

B9.

Degree of Compliance – Complete.

Explanation – Range is determined by declared precision except that range of a subscript expression is determined by array dimension. Run time exception conditions (on-conditions) are provided to detect truncation.

Scope of modifications – none.

B10.

Degree of Compliance - Partial.

Explanation - PL/I I/O operations are defined at a higher level than this requirement. Control of unique equipment must be specified through the ENVIRONMENT option which is implementation dependent.

Scope of modifications - this requirement can be satisfied by eliminating some current PL/I I/O operations, leaving the defined generic operations, and adding user type definitions to introduce new file and equipment types. PL/I EDIT and LIST I/O and sequential record I/O should be considered for inclusion.

B11.

Degree of Compliance - Partial.

Explanation - PL/I bit strings and the operations upon them provide the required functions, e.g. element predicate by SUBSTR (bitstring,n,1), union by A|B.

Scope of modifications - modifications to provide enumeration types are discussed in B6.

C1.

Degree of Compliance - Complete.

Explanation - PL/I does not impose this semantic restriction but an implementation is allowed to impose it.

Scope of modifications - all "Tinman" implementations of PL/I should choose this order of evaluation.

C2.

Degree of Compliance - Partial.

Explanation - PL/I has seven precedence levels. Explicit partheses are permitted but not required.

Scope of modifications - consider requiring explicit parentheses as indicated.

C3.

Degree of Compliance – Complete.

Explanation – expressions are permitted where both constants and variable references are allowed.

Scope of modifications – none.

C4.

Degree of Compliance – Complete.

Explanation – This holds for almost all cases where a constant is allowed. It does not apply to some uses of constants such as structure level numbers.

Scope of modifications – none.

C5.

Degree of Compliance – Complete.

Explanation – There is a consistent set of rules for parameters to user procedures and their declaration; and a superset of these rules applies to built-in functions, e.g. optional arguments to some built-in functions. There are no special operations applicable only to parameters. There are some references, e.g. to overlay defined arrays (iSUB), which cannot be used as arguments. See C7 for comment on parameter types.

Scope of modifications – subset the forms of reference which cannot be used as arguments. Ensure that parameters to definitions (see B1) are consistent with those for procedures.

C6.

Degree of Compliance – Partial.

Explanation – In current PL/I, formal and actual parameters must match or a valid conversion is compiled as part of the CALL. The modification described for B8 (eliminating implicit type conversion) provides complete compliance.

Scope of modifications – as described in B8.

C7.

Degree of Compliance - Partial.

Explanation - Current PL/I provides reference parameters and procedure parameters. Constants and optionally expressions are passed by reference to a copy, which prevents changes being seen by the caller. The function of exception parameters is provided by inheritance of ON-units.

Scope of modifications - Constant parameters could be introduced as specified. Procedure parameters might be omitted for simplicity, but if they are required there is a subset of the current PL/I language which is safe, checkable at compile-time and can be implemented efficiently. The ON-unit facilities might also be restricted to improve performance, but still provide the required function.

C8.

Degree of Compliance - Partial.

Explanation - PL/I has a GENERIC attribute although it does not work in exactly the way described.

Scope of modifications - The function described may be too complex to be included in the Tinman language. Consider simplifying or removing the GENERIC capability of PL/I. An alternative along the lines of that described could be introduced as a replacement.

C9.

Degree of Compliance - None.

Explanation - User procedures written in PL/I must have a fixed number of arguments. Some builtin functions accept a variable number of arguments.

Scope of modifications - None recommended.

D1.

Degree of Compliance - Partial

Explanation - the INITIAL attribute allows constants to be associated with variables; however, changing the value of that variable is allowed during program execution.

Scope of modifications - add an attribute to indicate that the value of the variable must not be changed.

D2.

Degree of Compliance - Complete.

Explanation - Scalar literals are provided for all atomic computational data types and are interpreted the same way, whether they appear in the source program or in program data.

Scope of modifications - none.

D3.

Degree of Compliance - Complete.

Explanation - PL/I satisfies all aspects of this requirement except that there is no explicit language to indicate run-time testing for initialization, although some implementations do perform this checking.

Scope of modifications - add mechanism for run-time testing for initialization in all "Tinman" implementations.

D4.

Degree of Compliance - Partial.

Explanation - range and stepsize is determined by declared precision and scale, and are thus integer powers of the number base.

Scope of modifications - see A4.

D5.

Degree of Compliance - Partial.

Explanation - PL/I complies for built-in types but does not currently allow defined types or enumerated types.

Scope of modifications - see E1 and E6 for modifications to allow defined types and enumerated types.

D6.

Degree of Compliance – Partial.

Explanation – PL/I complies except with respect to the safety mechanisms for using pointers.

Scope of Modification – Restrict PL/I name scope rules for pointers to eliminate pointers to data whose lifetime is more narrow than that of the pointer. Add capability to associate pointers with data types. These two modifications allow a safe implementation of PL/I pointers. Execution time checking of union type usage is required.

E1.

Degree of Compliance – None.

Explanation – PL/I provides limited capability for defining new data types and operations through structure declarations and procedure invocations. A significant expansion of this capability is necessary to meet this requirement.

Scope of Modification – Extend the concept of a procedure to cover a procedure that defines a data type. It will describe the structuring of the new data type and that structuring will be visible only within the procedure. The operations applicable to the new type will be internal procedures to the data defining procedure.

E2.

Degree of Compliance – None.

Explanation – Defined types are currently not allowed in PL/I.

Scope of modifications – the requirement of E2 is met by the modifications described for E1.

E3.

Degree of Compliance – Partial.

Explanation – PL/I allows explicit declarations but provides defaults if necessary to completey define attributes.

Scope of modifications – eliminate implicit and contextual declarations. It is assumed that selected attributes, e.g. INTERNAL/EXTERNAL can still be obtained by default rules.

E4.

Degree of Compliance – None.

Explanation – this capability is achieved by the modifications required to define new data types (E1).

Scope of modifications – no modifications other than E1 are required.

E5.

Degree of Compliance – None.

Explanation – see E1.

Scope of modifiations – the requirement of E5 is met by the modifications described for E1.

E6.

Degree of Compliance – None.

Explanation – PL/I bit strings do provide power sets over integers but new language is required to generalize this capability.

Scope of Modification – Provide ability to define a type by enumeration of literal names. (e.g. DCL 1 COLOR, 2 RED, 2 BLUE;).

Provide discriminated union, for the case where the actual type is instantiated at allocation and for the case where the actual type alters dynamically. (e.g. DCL 1 ITEM ONEOF, 2 A FIXED, 2 B FLOAT;).

Allow a bit string length to be determined by an enumeration type, so that it can be used as the power set for that type (e.g. DCL X BIT(COLOUR);).

E7.

Degree of Compliance – Complete.

Explanation – modifications required to comply with E6 will be defined such that they do not conflict with this requirement.

Scope of modifications – none.

E8.

Degree of Compliance - None.

Explanation - see E1.

Scope of modifications - the requirement of E8 is met by the modifications described for E1.

F1.

Degree of Compliance - Complete.

Explanation - PL/I makes this distinction between namescope and lifetime, e.g. STATIC INTERNAL. It is assumed that explicit control of allocation (BASED) is also required.

Scope of modifications - none.

F2.

Degree of Compliance - Partial.

Explanation - The PL/I rules of nested namescopes allow for the same name to be used for different variables. However the scope of an EXTERNAL name is the whole program. (Local aliases are permitted since the names of members of external structures are not themselves external.)

Scope of modifications - Limited access to type definitions will be part of the extension for E1.

More detailed control of EXTERNAL should be considered.

F3.

Degree of Compliance - Complete.

Explanation - PL/I namescope rules comply with this requirement.

Scope of modifications - none.

F4.

Degree of Compliance - Complete.

Explanation - PL/I has an extensive language-defined library of subroutines (built-in functions) as well as application oriented packages. The modifications required for E1 will provide additional capability.

Scope of modifications - none.

F5.

Degree of Compliance – Complete.

Explanation – PL/I source language components may be incorporated using the %INCLUDE capability. Compiled routines from any source language may be incorporated by the link-edit process.

Scope of modifications – all "Tinman" implementations must provide interface checking of routines when incorporated by the link-edit process.

F6.

Degree of Compliance – Complete.

Explanation – the %INCLUDE capability provides the necessary organization and control of stored data and routines.

Scope of modifications – consider addition of a capability for separate compilation of data structures.

F7.

Degree of Compliance – Complete.

Explanation – built-in functions such as TIME and DATE, List and Edit directed input/output, and storage management (ALLOCATE and FREE) are examples of generalized capabilities. The ENVIRONMENT option provides additional capabilities to specialize machine dependent functions.

Scope of modifications – the number of facilities of this kind should be carefully chosen, for language simplicity and easy compilation, to be things which are implementable in a straightforward way on most systems, without overhead for non-users.

G1.

Degree of Compliance – Partial

Explanation – PL/I has structured mechanisms for sequential, conditional, iteration, recursive control and exception handling. It does not have structured mechanisms for parallel processing and asynchronous interrupts.

Scope of Modifications – Add mechanisms for parallel processing (e.g. exclusive regions defined at compile time) and for asynchronous interrupts (e.g. events to be signalled).

Review the existing mechanisms for subsetting (e.g. a simpler DO) and for supersetting (e.g. addition of case selection).

G2.

Degree of Compliance – Partial.

Explanation – The PL/I GO TO is not restricted to local scope.

Scope of Modifications – Remove label variables and label parameters from PL/I. Consider restricting GO TO non-local labels to ON units where they provide the mechanism for exit from procedures through the dynamic calling structure (G7).

G3.

Degree of Compliance – Partial.

Explanation – PL/I allows a null ELSE clause to be written or omitted. PL/I does not have case selection and discriminated union.

Scope of Modifications – Add a case selection construct such as the SELECT statement for both value and type discrimination.

G4.

Degree of Compliance – Partial.

Explanation – PL/I allows entry only at the head, and does not impose excessive overhead in clarity or cost. PL/I does not allow termination to appear anywhere and does not require the iteration variable to be local.

Scope of Modifications – Provide a LEAVE statement (see latest IBM implementations of PL/I) to allow termination anywhere. Disallow explicit assignment to the iteration variable. Consider giving the iteration variable a special scope.

G5.

Degree of Compliance – Complete.

Explanation – Although PL/I does allow procedures internal to recursive procedures efficient implementations are practical.

Scope of Modifications – None.

G6.

Degree of Compliance – None.

Explanation – PL/I does not provide a parallel processing capability.

Scope of Modifications – Provide identifiers whose type is task (i.e. a parallel process). Provide statements to schedule and stop such tasks, and to define exclusive regions (see G1).

G7.

Degree of Compliance – Complete.

Explanation – The ON-unit capability of PL/I provides the necessary exception handling control structure. The concept of unrestricted dynamic descendence of PL/I ON-units introduces run-time inefficiencies for the non-user and should be modified.

Scope of Modifications – The best run-time efficiency is achieved when the ON-unit descendence is always known at compile-time. So we could consider subsetting the PL/I language to make this so.

If dynamic descendence is required – which is not certain – the run-time efficiency might still be made better than current PL/I by choosing a suitable subset.

G8.

Degree of Compliance – None.

Explanation – PL/I has no provision for real time, asynchronous control, beyond the functions to access time of day and date.

Scope of Modifications – In addition to extensions in G6, provide the ability to delay tasks, specify relative priorities, and accept asynchronous interrupts.

H1.

Degree of Compliance – Partial.

Explanation – PL/I meets this requirement except that abbreviation of keywords is allowed.

Scope of Modifications – Consider disallowing keyword abbreviations, or disallowing the long keywords that have abbreviations.

A

H2.

Degree of Compliance – Complete.

Explanation – The modifications required to comply with B4 will comply with this requirement.

Scope of Modifications – None.

H3

Degree of Compliance – Complete.

Explanation – PL/I requires 56 printable characters, and this character set will be used for the publication language. For programs written in smaller character sets (e.g. ASCII–64) there will be a mechanical translation (e.g. reserve the word AND for &) which will allow the program to be converted to the publication language, without losing any language facility.

Scope of Modifications – Provide the capability to do the necessary translation.

H4

Degree of Compliance – Partial.

Explanation – PL/I allows the underscore _ break character for identifiers but does not allow break characters in numeric constants. String constants may cross lines, but can also be split using the concatenate operator.

Scope of Modifications – Eliminate spanning lines for string constants. Consider allowing blanks after an underscore which breaks an identifier. Possibly allow blank or break character in numbers.

H5.

Degree of Compliance – Partial.

Explanation – PL/I does allow continuation of lexical units across lines. There are no restrictions on object characters contained in literal strings other than the limitations imposed by the input device, e.g. multipunching may be used to put control characters within literals.

Scope of Modifications – Disallow continuation of lexical units other than comments across lines.

H6.

Degree of Compliance – Complete.

Explanation – Keywords are not reserved in PL/I; the language can be parsed without. The keywords are informative and not usable in contexts where an identifier can be used. Builtin function names are not reserved, they are treated as identifiers.

Scope of Modifications – To make compilation easier, and programs more readable, make some keywords reserved (e.g. DCL, IF, THEN, ELSE, WHILE, etc.).

H7.

Degree of Compliance – Complete.

Explanation – PL/I comments meet the requirements except that they do not automatically terminate at end of line. It is felt that automatic termination at end of line constrains automatic reformatting of programs.

Scope of Modifications – None. The compiler should produce a warning message when a comment contains a semicolon or comment-starting delimiter.

H8.

Degree of Compliance – None.

Explanation – PL/I allows a single END statement to close multiple blocks.

Scope of Modifications – Disallow multiple closure.

H9.

Degree of Compliance – Partial.

Explanation – Function references, pseudo-variables and array element references are syntactically identical.

Scope of Modifications – The uniformity would be increased if we allowed user-defined pseudo-variables.

H10.

Degree of Compliance - Partial.

Explanation - PL/I uses the symbol = to mean both assignment and equality. Parenthesized arguments in procedure invocations indicate that a dummy argument is to be established. Otherwise, PL/I meets this requirement.

Scope of Modifications - Disallow parentheses around arguments in cases where they change the semantics. The assignment = could be changed to := but this would not be welcomed by those with PL/I experience.

I1.

Degree of Compliance - Partial.

Explanation - The PL/I language standard allows an implementation to choose certain parameters of the language such as maximum precision of fixed point data.

Scope of Modifications - The "Tinman" language definition can specify the value of these parameters for all implementations or allow user specification of these parameters.

I2.

Degree of Compliance - Complete.

Explanation - The PL/I language standard defines the effect of a program but not how the compiler is to implement it, leaving all such decisions to the compiler.

Scope of Modifications - Language could be introduced to specify such things (see J4 and J5). This should probably be written separately from the rest of the program.

I3.

Degree of Compliance - Complete.

Explanation - PL/I does not explicitly define compile time variables but does not prevent the translator from recognising computation that can be done at compile time.

Scope of Modifications - Ensure that there are simple rules which allow the coder to be sure that a particular selection will be done at compile time.

I4.

See I3.

I5.

Degree of Compliance – Partial.

Explanation – PL/I contains some features that could be programmed in terms of other more essential features, e.g. array assignment could be programmed with loops and scalar assignment.

Scope of Modifications – Examine such features carefully to determine whether they justify inclusion in the PL/I subset.

I6.

Degree of Compliance – Complete.

Explanation – No feature of PL/I prevents this approach, although the standard does not define limits.

Scope of Modifications – Define these limits for the "Tinman" language.

J1.

Degree of Compliance – Partial.

Explanation – PL/I has not succeeded in eliminating all features that have costs to the non-user of them.

Scope of Modifications – Subset PL/I to meet the requirement, for example restrict from the full generality of ON units.

J2.

Degree of Compliance – Complete.

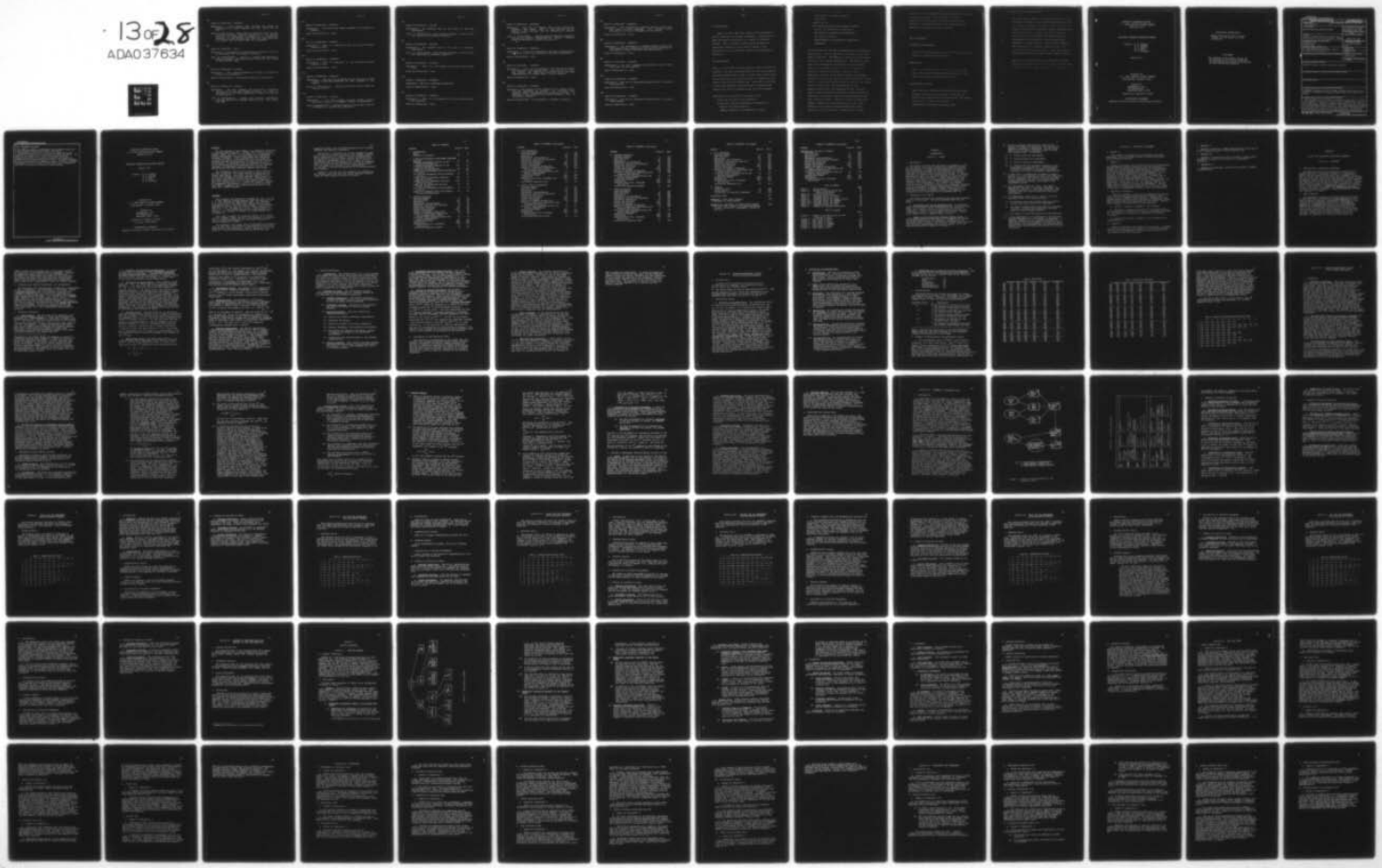
Explanation – Since the effect of the program may be real time dependent no translator can strictly meet this requirement. Additionally the PL/I definition permits translators some freedeom of choice, for example in the details of where conditions are raised. Optimizing and Checkout compilers might make different choices. The effect of the program would only be altered to the limited extent the language definition allows. We consider that this is the most that can be achieved without disallowing optimisation.

Scope of modification – None. Ensure that all translators correctly implement the language definition.

AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

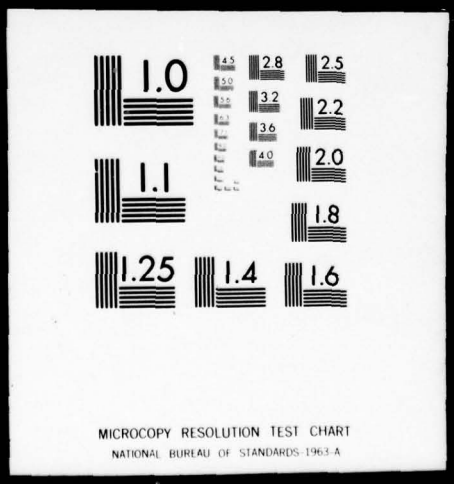
UNCLASSIFIED NL

13 OF 28
ADA037634



13 OF 28

ADA037634



J3.

Degree of Compliance – Partial.

Explanation – PL/I requires that non-PL/I code should be encapsulated in separate routines and link edited to the PL/I.

Scope of Modifications – Allow "builtin statements" where the verb represents a machine language operation and the compiler generates the appropriate accessing code to prepare the arguments for that operation e.g. ROTATE (A, J+1); /*if there is a machine code for cyclic shift of a register*/

J4.

Degree of Compliance – None.

Explanation – PL/I allows the specification of alignment but this is interleaved with the logical description.

Scope of Modifications – Provide a separate specification of physical mapping. This would specify the offset and length of each physical field.

J5.

Degree of Compliance – Partial.

Explanation – PL/I allows procedures to be open, but leaves the choice to the translator.

Scope of Modifications – Add explicit control of this choice.

K1.

Degree of Compliance – Partial.

Explanation – The PL/I standard does not require a particular operating system but some language facilities do assume an environment, in particular Input-Output assumes the existence of "data sets".

Scope of Modifications – Examine such features carefully to determine whether they justify inclusion in the PL/I subset (see B10).

K2.

Degree of Compliance – Complete.

Explanation – PL/I will permit either approach to integration of modules.

Scope of Modifications – None.

K3.

Degree of Compliance – Complete.

Explanation – There is experience of all the specified support applied to PL/I.

Scope of Modifications – None.

K4.

Degree of Compliance – Complete.

Explanation – There is experience of all the specified tools applied to PL/I.

Scope of Modifications – None.

K5.

Degree of Compliance – Complete.

Explanation – The coder can establish his own conventions within comments to make his assertions etc more apparent to the reader.

Scope of Modifications – Consider providing a second syntax for comment brackets.

L1.

Degree of Compliance – Partial.

Explanation – the PL/I language standard permits language extensions as long as the complete language is implemented.

Scope of Modifications – Disallow extensions except those required by "Tinman" e.g. machine code insertions.

L2.

Degree of Compliance – Partial

Explanation – The standard PL/I is too large to meet this requirement.

Scope of Modifications – Make the base language a subset of PL/I with extensions as indicated elsewhere.

L3

Degree of Compliance – Partial.

Explanation – The standard PL/I is too large to meet this requirement.

Scope of Modifications – Make the base language a subset of PL/I with extensions as indicated elsewhere.

L4.

Degree of Compliance – Complete.

Explanation – There are no features of PL/I that would prevent this.

Scope of Modfications – None.

L5.

Degree of Compliance – Complete.

Explanation – This is a permissive requirement.

Scope of Modifications – None.

L6.

Degree of Compliance – Complete.

Explanation – There are no features of PL/I that would prevent this approach.

Scope of Modification – None.

L7.

Degree of Compliance - Complete.

Explanation - PL/I has no features that would prevent this approach. Note however there is some conflict between rigidly defined messages and the flexibility of approach required by L8.

Scope of Modification - Consider defining suggested translator messages for "Tinman" implementations. Existing compilers prescribe experience to assist this process.

L8.

Degree of Compliance - Complete.

Explanation - It has been demonstrated that PL/I implementations can differ widely, from interpreters through to optimizers.

Scope of Modification - None.

L9.

Degree of Compliance - Complete.

Explanation - There are no features of PL/I that would prevent this approach. PL/I compilers have been written in PL/I. Note however that there may be pragmatic reasons why early implementations of "Tinman" adopt a diffent approach.

Scope of Modification - None.

M1.

Degree of Compliance - Complete.

Explanation - The features of PL/I included in the proposed subset have been successful in practice. The extensions being proposed follow established PL/I syntax and approach. The semantics of the extensions will follow established work in programming languages.

Scope of Modification - The extensions as described elsewhere.

M2.

Degree of Compliance – Complete.

Explanation – The existence of a formally defined standard PL/I will assist in this approach. The existing formal definitions in other meta languages are also useful.

Scope of Modification – None.

M3.

Degree of Compliance – Complete.

Explanation – The existence of a formally defined standard PL/I will assist in this approach, as will the existing textbooks, manufacturers manuals, and other published material.

Scope of Modification – None.

M4.

Degree of Compliance – Complete

Explanation – The PL/I language is an existing standard, which will assist in this approach.

Scope of Modification – None.

M5.

Degree of Compliance – Complete.

Explanation – PL/I has the necessary language features to support this approach.

Scope of Modification – None.

M6.

Degree of Compliance – Complete.

Explanation – PL/I has the necessary language features to support this approach.

Scope of Modification – None.

CPL1

## I. Introduction

CPL1 is a fully compatible subset of PL/I designed by CAP for real-time and systems programming. The language has been implemented successfully on a number of different systems. CPL1 is extremely simple and conventional in structure; probably its most unusual feature is its implementation as an extensible compiler, as described in Section II.

## II. Description

CPL1 is a restricted but fully compatible subset of the PL/1 language. The only incompatibility with PL/1 is the ability to insert machine code in-line. In some cases, quite severe restrictions have been imposed to make the language easy to compile into efficient object code. Since CPL1 is so close to PL/1, the language is described here in terms of the restrictions which it imposes on the full PL/1 language.

Restrictions fall basically into three areas:

- there are no built-in I/O operations.
- no run-time storage management is provided; this requires the elimination of:
  - BASED, CONTROLLED, and AUTOMATIC variables.

- -the ALLOCATE and FREE statements
- -recursion.
- -multitasking.

-restrictions which make the compiler simpler:

- -arrays are limited to one dimension.
- -no internal procedures with parameters
- -no structure arrays.
- -every ENTRY to a procedure has same set of parameters.

An added feature of the CPL1 implementation is the compiler extensibility. CPL1 is not really extensible in the usual sense of the word. The compiler is organized in two passes. The first pass is machine-independent. The output of this pass is an intermediate representation of the program comprising a sequence of macro calls. The second pass expands the macro calls, using a library of macro definitions specific to the object machine. A language extension is defined by adding new macro definitions to this library. Constructs from the language extension are ignored during the first pass, and processed during the macro-expansion pass. One advantage of this approach is that the code generated for language extension features is comparable in efficiency with the code generated for the basic language features. There are several disadvantages, however. Writing new macro definitions can be a very difficult task, requiring detailed knowledge of the strategies used for register assignment, etc., by the

existing macros. Moreover, an extension defined in this way is not transportable. The extension is really a modification to the machine-dependent portion of the compiler; this modification must be repeated for each machine on which the extension is desired.

## III. Discussion.

### A. Machine Independence.

The level of machine dependence is that of PL/1, except of course for machine-code insertions.

### B. Efficiency.

CPL1 is an extremely simple language with no dynamic allocation and no run-time checking performed at all. Thus the efficiency should be quite high.

### C. Real-Time Features.

Quite literally, there are no specifically real-time oriented features in this language. Even the PL/1 multitasking mechanism has been eliminated. All process management is assumed to be done by implementation-dependent supervisor calls.

D. System Programming Features.

The suitability of CPL1 as a system programming language is similar to that of PL/1, with the advantage that CPL1 is a smaller and simpler language, so that the size and speed of both compiler and object code are expected to be considerably better. However, certain of the CPL1 restrictions do seem to adversely affect its use in programming large systems.The separate compilation capability is quite limited. A very serious problem is the restrictions on multiple procedure entries and especially on internal procedures. These should make it significantly harder to produce highly modular software.

With these failings, and with no additional real-time facilities relative to PL/1, CPL1 is less suited to these applications than a number of existing PL/1 dialects.

TECHNICAL DOCUMENTARY REPORT

U. S. ARMY COMPUTER SYSTEMS COMMAND

USACSC-AT-76-11

# CANDIDATE LANGUAGES EVALUATION REPORT

Authors: B. M. Brosgol
R. E. Hartman
J. R. Nestor
M. S. Roth
L. M. Weissman

January 1977

Prepared for

U. S. ARMY COMPUTER SYSTEMS COMMAND

FORT BELVOIR, VIRGINIA 22060

Prepared by

INTERMETRICS, INC.
701 Concord Ave.
Cambridge, Mass. 02138

Contract No. DAHC-26-76-C-0006

DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

## DISPOSITION INSTRUCTIONS

Destroy this report when no longer needed. Do not return it to the originator.

## DISCLAIMER

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>USACSC-AT-76-11 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle)<br>CANDIDATE LANGUAGES EVALUATION REPORT | | 5. TYPE OF REPORT & PERIOD COVERED<br>Final Report for Period Oct. 1975 to Jan. 1977<br>6. PERFORMING ORG. REPORT NUMBER<br>IR-217 |
| 7. AUTHOR(s)<br>Benjamin M. Brosgol, Robert E. Hartman, John R. Nestor, Martin S. Roth, and Laurence M. Weissman | | 8. CONTRACT OR GRANT NUMBER(s)<br>DAHC26-76-C-0006 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Intermetrics, Inc.<br>701 Concord Ave.<br>Cambridge, Mass. 02138 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>6.27.25A/SX762725DY10/05/001 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>U.S. Army Computer Systems Command (CSCS-AT)<br>Fort Belvoir, Virginia 22060 | | 12. REPORT DATE<br>January 1977<br>13. NUMBER OF PAGES<br>531 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) | | 15. SECURITY CLASS. (of this report)<br>UNCLASSIFIED<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

programming language, high-order language, embedded computer applications, evaluation of programming languages, FORTRAN, COBOL, PL/I, TACPOL, CS-4, JOVIAL

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The main purpose of this report is to perform an evaluation of six High Order Languages -- TACPOL, CS-4, JOVIAL (J73/I), FORTRAN, COBOL, and PL/I -- with respect to the Department of Defense Requirements for High Order Computing Programming Languages ("Tinman"). These requirements were determined in a previous report to be consistent with the needed programming language characteristics for Army tactical and MIS applications. An evaluation tool is developed to assist in the investigation of the technical merit

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20 ABSTRACT (continued)

of the candidate languages, and management decision-making criteria are derived which take "non-technical" factors into account.

The fundamental result of this study is that, among the six HOLs considered, CS-4 comes closest to satisfying the Tinman requirements. The reason for the closeness of fit is that the language objectives receiving high priority in the Tinman (reliability, maintainability, readability) are also basic design goals of CS-4. This report recommends CS-4 as a suitable basis for a common DoD language. Because of fundamental shortcomings in each of the other five HOLs, this report recommends that these HOLs not be used as the basis for a common language.

TECHNICAL DOCUMENTARY REPORT

U. S. ARMY COMPUTER SYSTEMS COMMAND

USACSC-AT-76-11

# CANDIDATE LANGUAGES EVALUATION REPORT

January 1977

Authors: B. M. Brosgol
R. E. Hartman
J. R. Nestor
M. S. Roth
L. M. Weissman

Prepared for

U. S. ARMY COMPUTER SYSTEMS COMMAND

FORT BELVOIR, VIRGINIA 22060

Prepared by

INTERMETRICS, INC.
701 Concord Ave.
Cambridge, Mass. 02138

Contract No. DAHC26-76-C-0006

DA Project SX762725DY10

DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

ABSTRACT

The main purpose of this report is to perform an evaluation of six High Order Languages -- TACPOL, CS-4, JOVIAL (J73/I), FORTRAN, COBOL, and PL/I -- with respect to the Department of Defense Requirements for High Order Computing Programming Languages ("Tinman"). These requirements were determined in a previous report to be consistent with the needed programming language characteristics for Army tactical and MIS applications. An evaluation tool is developed to assist in the investigation of the technical merit of the candidate languages, and management decision-making criteria are derived which take "non-technical" factors into account.

The fundamental result of this study is that, among the six HOLs considered, CS-4 comes closest to satisfying the Tinman requirements. The reason for the closeness of fit is that the language objectives receiving high priority in the Tinman (reliability, maintainability, readability) are also basic design goals of CS-4. This report recommends CS-4 as a suitable basis for a common DoD language. Because of fundamental shortcomings in each of the other five HOLs, this report recommends that these HOLs not be used as the basis for a common language.

FOREWORD

This document was prepared by Intermetrics, Inc., for the U. S. Army Computer Systems Command, under the authority of U. S. Army Contract No. DAHC26-76-C-0006. This document incorporates CDRL Items A004, A005, and A006 of the above-referenced contract. The DA Project Number is SX762725DY10, Task Area is 05, and Work Unit is 001. Major Benjamin D. Blood, Jr., is the Contracting Officer's Representative for the Army. Dr. Benjamin M. Brosgol is Project Manager for Intermetrics.

This study evaluates the languages TACPOL, CS-4, JOVIAL (J73/I), FORTRAN, COBOL, and PL/I, with respect to the Department of Defense Requirements for High Order Computer Programming Languages ("Tinman").

The authors of this report wish to acknowledge the assistance of Judy Haigh, Jim Franklin, and Tonya Price during the preparation of the manuscript. Special thanks are due to Jim Felty for his many contributions during the editing and

production phases, and to Michelle Swanzy for her tireless "TECOing" and energetic assistance.

The material in Appendix III of this document is based on a review of an earlier version of the "Tinman" requirements, conducted by Dr. James S. Miller of Intermetrics, and sponsored by the Navy under Contract N00123-74-C-0634. In Chapter 4, paragraphs VIII.6, VIII.7, XI.1, and XI.3 are derived from work performed by Mr. Timothy A. Dreisbach under Air Force Contract F19628-76-C-0225, and paragraph VII.4 is based on a critique of CS-4 conducted by Dr. Benjamin M. Brosgol and sponsored by the above-mentioned Navy contract.

Chapters 3 through 8 of this document were produced on the Arpanet, using the TECO text editor and XOFF formatter under the TENEX operating system at USC-ISIC.

TABLE OF CONTENTS


| CHAPTER | | Section | Page |
|---|---|---|---|
| 1. | INTRODUCTION | | |
| | Scope | I | 1 |
| | Overview of Document | II | 3 |
| 2. | A TOOL FOR EVALUATING HIGH-ORDER LANGUAGES | | |
| | Approach | I | 5 |
| | Language-Independent Aspects of the Technical Evaluation | II | 13 |
| | Language-Independent Aspects of the Overall Evaluation | III | 19 |
| | Summary of Evaluation Tool | IV | 29 |
| | TACPOL and the Management Decision-Making Criteria | V | 34 |
| | CS-4 and the Management Decision-Making Criteria | VI | 37 |
| | JOVIAL and the Management Decision-Making Criteria | VII | 39 |
| | FORTRAN and the Management Decision-Making Criteria | VIII | 41 |
| | COBOL and the Management Decision-Making Criteria | IX | 44 |
| | PL/I and the Management Decision-Making Criteria | X | 47 |
| | Summary of Candidate HOLs with Respect to Cost Considerations | XI | 50 |
| 3. | TACPOL Evaluation | | |
| | Language Summary | I | 51 |
| | Data and Types | II | 60 |
| | Operations | III | 65 |
| | Expressions and Parameters | IV | 71 |
| | Variables, Literals and Constants | V | 76 |
| | Definition Facilities | VI | 80 |
| | Scopes and Libraries | VII | 83 |
| | Control Structures | VIII | 86 |
| | Syntax and Comment Conventions | IX | 90 |
| | Defaults, Conditional Compilation and Language Restrictions | X | 94 |
| | Efficient Object Representations and Machine Dependencies | XI | 98 |
| | Program Environment | XII | 101 |
| | Translators | XIII | 103 |
| | Language Definition, Standards and Control | XIV | 105 |
| | Conclusions Regarding TACPOL | XV | 107 |

TABLE OF CONTENTS (continued)


| CHAPTER | | Section | Page |
|---|---|---|---|
| 4. | CS-4 EVALUATION | | |
| | Language Summary | I | 110 |
| | Data and Types | II | 118 |
| | Operations | III | 122 |
| | Expressions and Parameters | IV | 127 |
| | Variables, Literals and Constants | V | 131 |
| | Definition Facilities | VI | 133 |
| | Scopes and Libraries | VII | 137 |
| | Control Structures | VIII | 141 |
| | Syntax and Comment Conventions | IX | 147 |
| | Defaults, Conditional Compilation and Language Restrictions | X | 152 |
| | Efficient Object Representations and Machine Dependencies | XI | 156 |
| | Program Environment | XII | 161 |
| | Translators | XIII | 162 |
| | Language Definition, Standards and Control | XIV | 164 |
| | Conclusions Regarding CS-4 | XV | 166 |
| 5. | JOVIAL (J73/I) EVALUATION | | |
| | *Language Summary* | I | 169 |
| | Data and Types | II | 177 |
| | Operations | III | 181 |
| | Expressions and Parameters | IV | 186 |
| | Variables, Literals and Constants | V | 190 |
| | Definition Facilities | VI | 193 |
| | Scopes and Libraries | VII | 196 |
| | Control Structures | VIII | 199 |
| | Syntax and Comment Conventions | IX | 204 |
| | Defaults, Conditional Compilation and Language Restrictions | X | 209 |
| | Efficient Object Representations and Machine Dependencies | XI | 212 |
| | Program Environment | XII | 215 |
| | Translators | XIII | 216 |
| | Language Definition, Standards and Control | XIV | 218 |
| | Conclusions Regarding JOVIAL | XV | 221 |

TABLE OF CONTENTS (continued)


| CHAPTER | | Section | Page |
|---|---|---|---|
| 6. | FORTRAN EVALUATION | | |
| | Language Summary | I | 224 |
| | Data and Types | II | 232 |
| | Operations | III | 239 |
| | Expressions and Parameters | IV | 245 |
| | Variables, Literals and Constants | V | 250 |
| | Definition Facilities | VI | 253 |
| | Scopes and Libraries | VII | 257 |
| | Control Structures | VIII | 260 |
| | Syntax and Comment Conventions | IX | 265 |
| | Defaults, Conditional Compilation and Language Restrictions | X | 269 |
| | Efficient Object Representations and Machine Dependencies | XI | 274 |
| | Program Environment | XII | 277 |
| | Translators | XIII | 279 |
| | Language Definition, Standards and Control | XIV | 282 |
| | Conclusions Regarding FORTRAN | XV | 284 |
| 7. | COBOL EVALUATION | | |
| | Language Summary | I | 289 |
| | Data and Types | II | 295 |
| | Operations | III | 299 |
| | Expressions and Parameters | IV | 306 |
| | Variables, Literals and Constants | V | 310 |
| | Definition Facilities | VI | 313 |
| | Scopes and Libraries | VII | 316 |
| | Control Structures | VIII | 319 |
| | Syntax and Comment Conventions | IX | 324 |
| | Defaults, Conditional Compilation and Language Restrictions | X | 329 |
| | Efficient Object Representations and Machine Dependencies | XI | 333 |
| | Program Environment | XII | 337 |
| | Translators | XIII | 339 |
| | Language Definition, Standards and Control | XIV | 342 |
| | Conclusions Regarding COBOL | XV | 344 |

TABLE OF CONTENTS (continued)


| CHAPTER | Section | Page |
|---|---|---|
| 8. PL/I EVALUATION | | |
| Language Summary | I | 348 |
| Data and Types | II | 359 |
| Operations | III | 364 |
| Expressions and Parameters | IV | 371 |
| Variables, Literals and Constants | V | 378 |
| Definition Facilities | VI | 381 |
| Scopes and Libraries | VII | 384 |
| Control Structures | VIII | 389 |
| Syntax and Comment Conventions | IX | 394 |
| Defaults, Conditional Compilation and Language Restrictions | X | 399 |
| Efficient Object Representations and Machine Dependencies | XI | 403 |
| Program Environment | XII | 407 |
| Translators | XIII | 409 |
| Language Definition, Standards and Control | XIV | 412 |
| Conclusions Regarding PL/I | XV | 414 |
| 9. SUMMARY | | |
| Evaluation Tool | I | 420 |
| Tinman Review | II | 422 |
| Suitability of Candidate Languages | III | 426 |
| LITERATURE CITED | | 437 |
| Appendix I, High Order Language Availability Questionnaire | | 439 |
| Appendix II, Department of Defense Requirements for High Order Computer Programming Languages "Tinman", Section IV, paragraphs A through M, June 1976 | | 445 |

TABLE OF CONTENTS (continued)


| CHAPTER | Section | Page |
|---|---|---|
| Appendix III, Review of "Tinman" | | |
| Introduction | I | 496 |
| Data and Types | II | 497 |
| Operations | III | 501 |
| Expressions and Parameters | IV | 505 |
| Variables, Literals and Constants | V | 509 |
| Definition Facilities | VI | 511 |
| Scopes and Libraries | VII | 512 |
| Control Structures | VIII | 513 |
| Syntax and Comment Conventions | IX | 516 |
| Defaults, Conditional Compilation and Language Restrictions | X | 518 |
| Efficient Object Representations and Machine Dependencies | XI | 519 |
| Program Environment | XII | 521 |
| Translators | XIII | 522 |
| Language Definition, Standards and Control | XIV | 523 |


LIST OF TABLES


| | | Page |
|---|---|---|
| Table I. | Rating Matrix | 16 |
| Table II. | Product of Rating Matrix and Application Vector | 18 |
| Table III. | Technical Evaluation | 31 |
| Table IV. | Language Vector for TACPOL | 34 |
| Table V. | Language Vector for CS-4 | 37 |
| Table VI. | Language Vector for JOVIAL (J73/I) | 39 |
| Table VII. | Language Vector for FORTRAN | 41 |
| Table VIII. | Language Vector for COBOL | 44 |
| Table IX. | Language Vector for PL/I | 47 |
| Table X. | Summary of Language Evaluations | 434 |


LIST OF FIGURES


| | | Page |
|---|---|---|
| Figure 1. | Stages in the Application of the Evaluation Tool | 30 |
| Figure 2. | Data Types in TACPOL | 52 |
| Figure 3. | Data Types in CS-4 | 111 |
| Figure 4. | Data Types in JOVIAL | 171 |
| Figure 5. | Data Types in FORTRAN | 225 |
| Figure 6. | Data Types in COBOL | 290 |
| Figure 7. | Data Types in PL/I | 349 |

# CHAPTER 1

# INTRODUCTION

## Section I. SCOPE

### 1. Background.

The present contract from the Army to Intermetrics calls for two general activities, occurring in sequential phases: the determination of language requirements for an Army standard programming language, and the evaluation of various high-order languages (HOLs) to determine the most suitable candidate for Tactical Data Systems applications. A Language Requirements Report [7] has been prepared in connection with the first of these activities. A major conclusion of the report is that there is no inconsistency between the basic language requirements for "general purpose" as opposed to tactical or MIS programming. As a result, the "needed characteristics" discussed in Section IV, paragraphs A through J, of the Department of Defense Requirements for High Order Computer Programming Languages "Tinman" (June 1976) [16], which pertain mostly to "general purpose" programming, are acceptable as the requirements for a standard Army HOL.

### 2. Purposes of This Report.

This report contains the results of the language evaluation phase of the contract. The work performed falls into two areas.

a. Development and Use of Evaluation Tool. An important aspect of this study is the formulation of an "evaluation tool" which can be used to assist in the process of selecting a common HOL. Chapter 2 of this report presents the tool and illustrates its use in the evaluation of six HOLs: TACPOL, CS-4, JOVIAL (J73/I), FORTRAN, COBOL, and PL/I.

b. Support of DoD High Order Language Working Group. A second major purpose of this report is to support the efforts of the HOL Working Group by supplying a comparison of the above-mentioned six HOLs to the Tinman's "needed characteristics." The following subparagraphs describe the format of the comparisons presented in Chapters 3 through 8.

(1) For each language characteristic listed in the Tinman, the "degree of compliance" for each of the candidate languages is determined. The following symbols will be used to represent levels to which each language meets a requirement:

  (a) T: Totally meets the requirement.

  (b) P: Partially meets the requirement.

  (c) F: Fails to meet the requirement.

  (d) U: Unknown from the available documents if the requirement is satisfied (e.g., the requirement is strictly dependent on the operating system, libraries, or is only a management consideration).

(2) In some cases, a combination of symbols is appropriate; e.g., "PT" represents nearly total compliance, and "FU" indicates that the HOL fails to meet part of the requirement and that it is unknown from the language definition whether other parts of the requirement are met.

(3) For requirements which are "met", this report demonstrates how this is achieved (typically, the presence or absence of a particular facility), and indicates when the solution conflicts with other requirements.

(4) For requirements which are not "met" or are only partially "met", this report provides:

  (a) An analysis of why the language does not fulfill or only partially fulfills the requirement.

  (b) The scope of modifications that would be necessary to bring a language into compliance, and their impact on other language features.

  (c) The impact of such modifications on implementation.

(5) If the requirement is not addressed in the provided language documentation, this report so indicates. In addition, features of the candidate languages are identified which are not needed to satisfy the Tinman requirements, and recommendations are presented as to whether these features should be retained or possibly eliminated.

## Section II. OVERVIEW OF DOCUMENT

1. Chapter 1.

This report is divided into nine chapters and three appendices, with the current chapter providing background and introductory material.

2. Chapter 2.

Chapter 2 describes the evaluation tool and the management decision-making criteria relevant to its use (Sections I through IV) and presents an illustration of the application of the tool to the candidate HOLs (Sections V through XI). Since the Tinman's characteristics in paragraphs K, L, and M do not correspond directly to language features, the evaluation of the HOLs' "technical merit" is based just on comparisons with Tinman paragraphs A through J. In the derivation of the technical merit scores, numeric ratings are given which refine the "T", "P", "F", and "U" degrees of compliance. A rating of 0.9 or 1.0 indicates that the language (nearly) totally satisfies the requirement. Partial compliance is reflected by scores of 0.6, 0.7, or 0.8; lower scores reveal a fundamental failure to meet the requirement.

3. Chapters 3 through 8.

Chapters 3 through 8 (which do not depend on the evaluation tool) contain detailed evaluations of TACPOL, CS-4, JOVIAL (J73/I), FORTRAN, COBOL, and PL/I, respectively, and recommendations concerning the suitability of modifying these HOLs to bring them into compliance with the Tinman. Each of these chapters is divided into fifteen sections.

a. Section I contains a summary of the language.

b. Sections II through XIV consist of the evaluations of the language with respect to the Tinman's paragraphs A through M. This material is organized as described in paragraph I.2b above.

c. Section XV presents a summary of the evaluation and recommendations.

4. Chapter 9.

Chapter 9 summarizes the results of this study. A synopsis of the evaluation tool, an outline of a review of the Tinman, and a set of recommendations concerning the suitability of the candidate HOLs are presented.

5. Appendix I.

Appendix I contains a sample questionnaire on HOL availability, proposed as part of the evaluation tool.

6. Appendix II.

Appendix II contains the set of Tinman's "needed characteristics" as found in [16, Section IV, paragraphs A through M].

7. Appendix III.

Appendix III provides a review of the Tinman's "needed characteristics."

# CHAPTER 2

# A TOOL FOR EVALUATING HIGH-ORDER LANGUAGES

## Section I. APPROACH

### 1. Difficulties in Evaluating Languages.

Language evaluation is intrinsically a difficult and complex activity: a computer language is an intangible entity, and the factors which enter into a judgment of the merit of a HOL tend to be subjective and sometimes conflicting. The problem is that there is no such thing as a "good" language in absolute terms--a HOL which is appropriate in one environment may prove to be exactly the opposite in another. Instead, one should only speak of the merit of a language in relative terms: with respect to an application area for which the HOL is to be used, and with respect to a computing environment (hardware equipment, personnel, financial resources) in which the language is to appear.

### 2. Basic Approach.

The approach to language evaluation taken here attempts to cope with these difficulties by partitioning them into more manageable components (specificably addressed to application area and computing environment), and by making visible, in a quantitative fashion, the individual judgments which enter into the language evaluation process. This is not to understate the problem of accurately estimating these quantitative values; however, it is felt that the tool proposed will provide management with a handle on the basic factors which influence language selection. Moreover, if resources are available for deriving the

numeric values via an approach such as the Delphi method[1], the accuracy of the estimates will be increased. It should be emphasized, though, that we are not advocating the selection of one HOL rather than another on the sole basis of a mechanical choice of the HOL with the highest score. Rather, the evaluation tool described here is intended to serve primarily as a comprehensive guide for orderly thinking.

3. Technical and Overall Evaluations: An Introduction.

The evaluation tool proposed here is basically a methodology, or set of procedures, to be conducted in two stages. First, a technical evaluation (TE) will produce a score for the language's technical merit in meeting the requirements of a particular application area -- in this report the languages under consideration are TACPOL, CS-4, JOVIAL (J73/I), FORTRAN, COBOL, and PL/I, and the application area is that of a standard HOL for Army tactical and MIS systems. This score will then be used in the second stage in the overall evaluation (OE) of the language. The OE will take into account the computing environment in which the language will be used (viz., Army sites) via a set of management decision-making criteria.

4. Technical Evaluation.

a. Rating Matrix. The main part of the approach is the derivation of a Rating matrix, R, whose rows correspond to language goals (e.g., reliability, efficiency of object code, etc.). The value $R_{ij}$ will be a rating of how the $i^{th}$ feature contributes toward the $j^{th}$ goal, and will be in the range of 0 to 10. For example, a feature like TACPOL's CODE statement, which allows a user to descend into assembly language, would have a high rating with respect to object code efficiency, but would rate poorly with respect to the goals of program reliability and transportability.

---

[1]According to the Delphi method, the members of a group of independent experts offer judgments on specific questions on two or more successive occasions. At each iteration, the amount of agreement on the chosen values is made available to the whole group, along with a list of reasons for the choices. Each group member is free to use this information in his reevaluation. This permits access by each member to the knowledge pool of the whole group, and results in a set of agreed-upon numeric values.

b. Language- and Application-Independence. It should be noted that the matrix R is both language-independent and application-independent. Language- and application-dependent vectors will be multiplied by R to obtain a score for a particular language with respect to a given application; this is described in more detail below.

c. Derivation of Rating Matrix. Three relevant issues which arise are: (1) How is the set of language features determined? (2) How is the set of language goals decided? (3) How are the ratings $R_{ij}$ derived? With respect to language features, it was noted earlier that previous work under this contract [7] concluded that the Tinman's "needed characteristics" (Sections A through J) are suitable as the set of required features for a standard Army HOL. Thus these 78 characteristics will correspond to the rows of the matrix. The goals were determined by examining the program development process (problem specification, design, coding, debugging, maintaining) and identifying the most important objectives for a HOL at the various stages: learnability, power, efficiency, reliability, maintainability, and transportability. The entries in the Rating Matrix were derived by Intermetrics and reviewed by USACSC.

d. Language Vector. The Rating Matrix, R, say with m rows and n columns (here m=78 and n=6), is used in conjunction with two vectors for the purpose of the Technical Evaluation. The Language vector, L, is a row vector with m components (the number of language features). For each candidate language C, there is a Language vector $L^C$; the $i^{th}$ element in $L^C$ is a weighting between 0.0 and 1.0 (inclusive) which reflects how well language C satisfies the $i^{th}$ Tinman requirement. It should be noted that the product $L^C * R$ is a row vector with n components; its $j^{th}$ element is a score between 0 and 780 giving the overall rating of language C with respect to the $j^{th}$ goal.

e. Application Vector. The Application vector, A, is a column vector with n components (the number of language goals). For any given application area, U, there is an application vector $A^U$ satisfying the following conditions:

(1) $A_j^U \geq 0$ for $j = 1, \ldots, n$

(2) $\sum_{j=1}^{n} A_j^U = 1.0$

Intuitively, $A_j^U$ is the relative importance of the $j^{th}$ goal for the application U. For example, for tactical applications, the goal of object code efficiency might receive a higher weight than the goal of transportability. The reason for requiring the weights in $A^U$ to sum to 1.0 is that goals frequently overlap (e.g., learnability, reliability, maintainability), and allowing non-normalized values in $A^U$ would result in overemphasis of such goals (i.e., effectively, the same goal would be counted several times).

f. Intermediate Vector. The product $R * A^U$, denoted $IV^U$, is a column vector, language-independent, with m components. Its $i^{th}$ element is a score between 0 and 10 which reflects how well the $i^{th}$ Tinman requirement contributes toward application area U.

g. Technical Merit. The product $L^C * IV^U$ yields a scalar value which reflects the technical merit of language C with respect to application area U. The maximum value for this product is obtained when C scores 1.0 for each entry in $IV^U$; we denote this summation by $IV^U_*$; i.e., $IV^U_* = \sum_{i=1}^{m} IV^U_i$ where m is the number of features. Thus the range for $L^C * IV^U$ is 0 through $IV^U_*$. Since, for convenience of reference as well as for potential use in the overall evaluation we are interested in the range 0 through 100, we define the technical merit of language C, with respect to application area U, to be $100 * (L^C * IV^U)/IV^U_* = K(U) * (L^C * IV^U)$ where the transformation constant $K(U) = 100/IV^U_*$.

h. Absolute Requirements. For certain applications, a number of requirements are "absolute" in that a language lacking these requirements cannot be used. As an example, for list processing applications the following requirement (from D6 in the "Tinman") is absolute: "The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure." For tactical applications, the "Tinman" requirement J3 (Machine Language Insertions) is absolute. Thus, for the purposes of this report, any language which does not satisfy J3 would be excluded from further consideration. FORTRAN and PL/I -- as described by their reference documents as opposed to their implementations -- fall into this category.

5. Overall Evaluation.

a. Introduction. The technical merit of a given language with respect to a particular application area is derived in the Technical Evaluation stage; the Overall Evaluation takes this score into account as well as other factors specific to the computing environment in which the language is to be used. These factors interact so as to affect the cost of producing programs in the given language. The overall evaluation attempts to account for these effects and to help estimate the costs involved for any candidate HOL.

b. Breakdown of Costs. The non-hardware expenses incurred during the development of a software system can be classified into the following categories:

(1) Language acquisition. This entails obtaining a language processor, support tools and documentation so that the system can be programmed.

(2) Programmer training. Available or new personnel may have to be trained to use the language selected.

(3) System development. This area entails the following activities:

(a) Defining the system performance requirements.

(b) Designing the system.

(c) Coding the system in the given language(s).

(d) Testing, debugging, and verifying the software.

(e) Maintaining and upgrading the system, perhaps in response to changes in the performance requirements.

(f) Transporting the system perhaps to new hardware configurations.

(4) Indirect expenses. This category includes expenses resulting from not meeting the system performance requirements -- e.g., costs stemming from late or incorrect software.

c. Management Decision-Making Criteria. The actual costs associated with b(1) through b(4) above depend on a variety of factors. Some of these factors relate directly to the language chosen for system implementation; we term these factors "management decision-making criteria" since they may be used by management to help guide the language selection process. The set of criteria considered in this report are technical merit; compiler, support tool, and documentation availability; standardization status; industry support; and availability of trained programmers.

d. Outline of Overall Evaluation. The approach suggested here is to utilize a set of values for each candidate language's compliance with the management decision-making criteria. This set will be applied in order to determine the impact of these criteria upon the various costs associated with software system development. That is, for candidate language C, application area U and computing environment E, there will be a Language Management vector $LM^{C,U,E}$ whose $i^{th}$ component is a rating of how well language C meets the $i^{th}$ decision-making criterion; $0 \leq LM_i^{C,U,E} \leq 100$ for each i. The cost associated with the $j^{th}$ "expense component" (language acquisition, programmer training, system development, indirect expenses) will then depend on the values in $LM^{C,U,E}$. Let $X_j^{C,S}$ be the expense associated with the $j^{th}$ cost component for candidate language C and software system S; then $X_j^{C,S}=f_j(LM^{C,U,E},S)$ for some function $f_j$. If the functions $f_j$ could be approximated, the user of the evaluation tool could obtain estimates on the effect of the language upon system cost; these estimates could then be used to assist in the process of HOL selection.

## 6. Limitations of the Evaluation Tool.

The evaluation tool described in this report has good potential for lending assistance during the HOL selection process, and perhaps its greatest benefit emerges from the way it induces an organized and orderly investigation into the factors which influence the choice of a language. Nevertheless, there are several limitations on the applicability of the tool; these are outlined in the following paragraphs.

a. Numeric Approach. The problems associated with obtaining quantitative estimates for the various components of the tool are severe. Techniques such as the Delphi method are helpful; however, a fair amount of research is needed in order to obtain an information baseline to guide the participants in these techniques. In addition, certain aspects of the Tinman document caused complications in the derivation of numeric scores during the Technical Evaluation stage for this report. One problem is the absence of precise definitions for the terms used in the various "needed characteristics". Although it might be argued that the provision of such definitions would overemphasize individual features at the expense of general requirements, it is nevertheless true that the ambiguities in the Tinman cause many of the scores to depend on specific interpretations. (A review of the Tinman, including illustrations of ambiguity, is given in Appendix III of this report.) Another problem in using the Tinman is that most of the "needed characteristics" are not single requirements. Typically a characteristic comprises a set of requirements, and often these constituents vary in their importance; this makes it a difficult decision to compose a single score for the HOL's compliance with the characteristic.

b. "New" Languages. In the application of the tool, the technical evaluation is based solely on the defining document for the language, while the overall evaluation takes into account the history of the language's use. There are problems, however, when the defining document introduces revisions to an existing language (in this report, FORTRAN, COBOL and PL/I illustrate this situation). The difficulty is that the language being technically evaluated may be considerably different from the language actually in use. In this report we account for such circumstances by describing the degree of difference between the languages as defined and as implemented; the more significant this difference, the lower the non-technical ratings we should use during the overall evaluation.

c. Modifiability Constraints. A basic aspect of the tool is that each candidate language is evaluated "statically" -- i.e., with respect to its closeness to the requirements but not with respect to the ease or difficulty of making modifications to bring it into compliance. In other words, the tool as described here is applicable to determining the "best fit" to the requirements, but is not directly applicable to determining how to modify the language so

that it meets the requirements. It should be emphasized that discussions concerning the scope of modificaations needed to bring the various candidate HOLs into compliance with the Tinman are supplied in the subsequent chapters of this report; however, these are not utilized in the evaluation tool. An extension of the tool to take such data into account appears to be a fruitful area for further research.

## Section II. LANGUAGE-INDEPENDENT ASPECTS OF THE TECHNICAL EVALUATION

### 1. Introduction.

To carry out the Technical Evaluation stage of the evaluation of language C for application area U, we must derive the language vector $L^C$, the Rating matrix R, and the Application Vector $A^U$. This section presents $A^U$ and R, which are independent of the language being evaluated; the vectors for the various candidate HOLs are given in Sections V through X.

### 2. Application Vector.

a. Selection of Language Goals. The application vector $A^U$ consists of non-negative numeric weightings which sum to 1.0; the $j^{th}$ element in $A^U$ reflects the relative importance of the $j^{th}$ goal in application area U. The number and variety of the goals which are relevant to high-order languages are quite large, and choosing from these goals a set to be used in the language evaluation is not a straightforward decision. In making the selection of goals, we attempted to avoid choosing ones which overlapped (however, some redundancy is inevitable due to the interrelationships among language objectives). In addition, we were guided by the use to be made for the goals -- viz., in the rating matrix and application vector. Thus, the ability to estimate how well a language feature contributed toward meeting a goal, or how appropriate a goal was in a given application area, were important in guiding the selection. The set of goals to be used in this study are: learnability, power, efficiency, reliability, maintainability, transportability. As mentioned earlier in the Chapter (paragraph I.4.c), these goals are applicable at the various stages of the program development process (problem specification, design, coding, debugging, maintaining). Learnability is relevant at all stages; power is most applicable during design and coding; efficiency and reliability are relevant in coding, debugging, and maintaining; maintainability is applicable during debugging and maintaining; transportability is relevant during program maintenance.

b. Definition of Language Goals.

(1) Learnability. The goal of learnability encompasses simplicity, uniformity of notation, and consistency of rules. A learnable language will have a short, clear defining document and will make possible the production of a small fast compiler.

(2) Power. Power implies both generality (i.e., applicability to a diversity of programming areas) and problem-oriented high-level notation. Extensibility in a HOL contributes toward power.

(3) Efficiency. The efficiency of a language measures how well the code generated by a good compiler will compare (in storage and speed) to that produced by a good assembly language programmer. With respect to the degree to which a given HOL feature contributes toward efficiency, the important consideration is the quality of the code which can be produced, and whether runtime overhead is implied.

(4) Reliability. By "reliability" we mean the ability of a language to permit and facilitate the writing of demonstrably correct programs. Compile-time checking, program test and debug features, and facilities which allow formal proofs of correctness, are some of the factors contributing toward this goal.

(5) Maintainability. Program maintainability includes the goals of readability and ease of constructing large systems. Facilities which support maintainability range from source listing formatters to language features such as data abstraction which help to localize the program changes in an evolving system.

(6) Transportability. By "transportability" we mean the ability to move a program intact from one host-target-machine environment to another and obtain the same result (except for possible differences in the precision of arithmetic values caused by differences in machine word lengths). Transportability implies an absence of "undefineds" in the language specification.

c. Weightings for the Application Vector Components. The weightings derived for the various goals, when applied to the application area of a standard Army HOL for tactical systems, are as follows:

| | |
|---|---|
| Learnability: | .10 |
| Power: | .15 |
| Efficiency: | .20 |
| Reliability: | .25 |
| Maintainability: | .20 |
| Transportability: | .10 |

3. Rating Matrix.

The Rating matrix has 78 rows (the number of "needed characteristics" in Section IV of the Tinman) and 6 columns (the number of goals). Entries in the matrix are values between 0 and 10, with the following interpretations:

| Weighting value | Explanation |
|---|---|
| 10 | The feature is relevant to the goal and strongly helps attain the goal. |
| 7.5 | The feature is relevant to, and does not conflict with, the goal. |
| 5 | The feature has no effect on either attaining or defeating the goal. |
| 2.5 | The feature is relevant to, and conflicts with, the goal. |
| 0 | The feature is relevant to the goal and strongly helps defeat the goal. |

Table I displays the Rating matrix; the three-character codes along the left side correspond to the individual "needed characteristics" in the Tinman.

4. Product of Rating Matrix and Application Vector.

a. The intermediate result $IV^U = R * A^U$ in the technical evaluation is a 78x1 column vector. The $i^{th}$ entry is a measure of the suitability of the $i^{th}$ Tinman requirement with respect to the application area of a standard Army HOL; it is a value between 0 and 10. The entries in this vector are displayed in Table II, and these values reflect directly the desirability of the various Tinman characteristics with respect to the given application area. Here we see that type definitions (E5) and data defining mechanisms (E6)

TABLE I. RATING MATRIX

| | LEARNABILITY | POWER EFFICIENCY | RELIABILITY | MAINTAINABILITY | TRANSPORTABILITY |
|---|---|---|---|---|---|
| A01 | 7.5 | 8.5 | 5.0 | 10.0 | 6.5 | 6.5 |
| A02 | 7.5 | 9.5 | 5.5 | 9.5 | 7.5 | 7.5 |
| A03 | 6.0 | 5.5 | 6.0 | 7.5 | 6.5 | 9.5 |
| A04 | 3.5 | 8.0 | 4.5 | 4.5 | 4.5 | 8.5 |
| A05 | 6.0 | 6.0 | 5.0 | 5.5 | 5.5 | 6.0 |
| A06 | 9.0 | 9.5 | 4.0 | 9.5 | 8.5 | 6.5 |
| A07 | 7.0 | 9.5 | 3.0 | 9.5 | 7.5 | 7.5 |
| B01 | 8.5 | 8.5 | 6.5 | 8.0 | 7.5 | 7.0 |
| B02 | 8.5 | 7.5 | 6.5 | 8.0 | 7.5 | 7.0 |
| B03 | 8.5 | 8.5 | 5.0 | 8.0 | 7.5 | 7.0 |
| B04 | 8.0 | 7.0 | 7.0 | 7.5 | 7.5 | 6.5 |
| B05 | 6.0 | 5.0 | 3.5 | 8.0 | 6.5 | 5.0 |
| B06 | 8.5 | 8.5 | 6.5 | 8.0 | 8.0 | 5.5 |
| B07 | 3.0 | 9.0 | 7.5 | 3.0 | 3.5 | 5.5 |
| B08 | 9.0 | 6.0 | 6.5 | 9.5 | 9.0 | 5.5 |
| B09 | 7.0 | 6.0 | 4.0 | 7.5 | 7.5 | 5.5 |
| B10 | 7.5 | 9.5 | 5.0 | 7.5 | 7.5 | 9.0 |
| B11 | 5.5 | 7.5 | 7.0 | 6.0 | 6.0 | 5.5 |
| C01 | 7.5 | 5.5 | 8.0 | 7.5 | 7.5 | 7.5 |
| C02 | 8.5 | 4.0 | 5.0 | 8.5 | 8.5 | 5.0 |
| C03 | 8.5 | 8.5 | 4.5 | 8.5 | 8.5 | 5.0 |
| C04 | 6.5 | 8.5 | 5.5 | 7.5 | 7.5 | 5.5 |
| C05 | 8.0 | 7.5 | 5.0 | 8.0 | 8.0 | 5.5 |
| C06 | 9.0 | 7.5 | 4.5 | 9.0 | 9.0 | 5.5 |
| C07 | 8.5 | 9.0 | 6.0 | 8.5 | 8.5 | 5.5 |
| C08 | 7.0 | 8.5 | 5.5 | 7.0 | 7.0 | 5.5 |
| C09 | 7.0 | 8.5 | 5.5 | 7.0 | 7.0 | 5.5 |
| D01 | 8.0 | 8.5 | 5.5 | 8.0 | 8.0 | 5.5 |
| D02 | 8.5 | 8.0 | 5.5 | 8.5 | 8.5 | 7.5 |
| D03 | 7.0 | 9.0 | 5.5 | 7.5 | 7.5 | 5.5 |
| D04 | 8.0 | 8.0 | 4.5 | 8.5 | 8.5 | 5.5 |
| D05 | 9.0 | 8.5 | 5.0 | 8.5 | 8.5 | 6.0 |
| D06 | 4.5 | 5.5 | 9.5 | 4.5 | 4.5 | 6.0 |
| E01 | 6.5 | 10.0 | 6.0 | 7.5 | 8.0 | 6.0 |
| E02 | 9.0 | 7.5 | 6.0 | 9.0 | 9.0 | 6.5 |
| E03 | 9.0 | 5.0 | 5.0 | 9.0 | 9.0 | 5.0 |
| E04 | 5.5 | 8.0 | 5.0 | 5.5 | 5.5 | 5.0 |
| E05 | 9.0 | 9.5 | 5.5 | 9.5 | 9.5 | 6.5 |
| E06 | 9.0 | 9.5 | 5.5 | 9.5 | 9.5 | 6.5 |
| E07 | 9.0 | 5.0 | 5.0 | 9.0 | 9.0 | 5.0 |
| E08 | 9.0 | 9.5 | 5.5 | 9.0 | 9.0 | 5.5 |

TABLE I. RATING MATRIX (CONTINUED)

| | LEARNABILITY | POWER EFFICIENCY | RELIABILITY | MAINTAINABILITY | TRANSPORTABILITY |
|---|---|---|---|---|---|
| F01 | 7.5 | 9.0 | 5.0 | 7.5 | 7.5 | 5.0 |
| F02 | 9.0 | 8.5 | 5.0 | 9.5 | 9.5 | 5.5 |
| F03 | 8.5 | 7.0 | 7.5 | 8.5 | 8.5 | 5.0 |
| F04 | 7.5 | 8.5 | 6.5 | 9.0 | 9.5 | 6.0 |
| F05 | 7.5 | 9.0 | 5.5 | 8.5 | 9.5 | 5.5 |
| F06 | 7.5 | 9.0 | 5.0 | 8.5 | 9.5 | 5.5 |
| F07 | 9.0 | 6.0 | 4.0 | 9.0 | 9.0 | 9.5 |
| G01 | 7.0 | 9.0 | 4.5 | 9.0 | 9.0 | 8.0 |
| G02 | 7.5 | 8.0 | 8.0 | 6.5 | 6.5 | 5.0 |
| G03 | 8.5 | 8.0 | 7.5 | 8.5 | 8.5 | 5.0 |
| G04 | 8.5 | 8.0 | 7.5 | 8.5 | 8.5 | 5.0 |
| G05 | 6.0 | 9.5 | 4.5 | 6.0 | 6.0 | 5.0 |
| G06 | 7.5 | 9.5 | 3.5 | 7.5 | 7.5 | 8.5 |
| G07 | 7.5 | 9.5 | 4.5 | 7.5 | 7.5 | 8.5 |
| G08 | 7.5 | 9.5 | 4.5 | 7.5 | 7.5 | 8.5 |
| H01 | 9.0 | 5.0 | 5.0 | 9.0 | 9.0 | 5.0 |
| H02 | 9.0 | 4.0 | 5.0 | 9.0 | 9.0 | 5.0 |
| H03 | 5.5 | 5.0 | 5.0 | 5.5 | 5.5 | 9.0 |
| H04 | 7.5 | 5.0 | 5.0 | 7.5 | 7.5 | 5.0 |
| H05 | 7.5 | 5.5 | 5.0 | 7.5 | 7.5 | 5.0 |
| H06 | 8.0 | 5.0 | 5.0 | 8.0 | 8.0 | 5.0 |
| H07 | 6.0 | 5.0 | 5.0 | 6.0 | 6.0 | 5.0 |
| H08 | 8.0 | 5.0 | 5.0 | 8.0 | 8.0 | 5.0 |
| H09 | 7.5 | 5.0 | 5.0 | 7.5 | 7.5 | 5.0 |
| H10 | 8.0 | 5.0 | 5.0 | 8.0 | 8.0 | 5.0 |
| I01 | 8.0 | 5.0 | 5.0 | 8.0 | 8.0 | 7.5 |
| I02 | 7.5 | 5.0 | 5.5 | 7.5 | 7.5 | 6.5 |
| I03 | 6.5 | 7.5 | 6.0 | 6.5 | 6.5 | 6.0 |
| I04 | 6.5 | 8.0 | 7.5 | 6.5 | 6.5 | 7.0 |
| I05 | 8.0 | 7.5 | 4.5 | 8.0 | 8.0 | 6.5 |
| I06 | 5.0 | 5.0 | 5.0 | 5.0 | 5.0 | 8.5 |
| I07 | 5.5 | 5.0 | 5.0 | 5.0 | 5.0 | 4.5 |
| J01 | 5.0 | 4.5 | 9.5 | 5.0 | 5.0 | 5.0 |
| J02 | 7.5 | 5.0 | 8.0 | 7.5 | 7.5 | 5.0 |
| J03 | 3.5 | 9.5 | 10.0 | .5 | .5 | .0 |
| J04 | 5.0 | 5.0 | 9.5 | 5.0 | 5.0 | 4.5 |
| J05 | 5.0 | 5.0 | 9.0 | 5.0 | 5.0 | 5.0 |

score most highly (about 8.4), while the machine language insertions requirement (J3) emerges as least desirable (4.0). This latter conclusion may seem surprising, in light of the high dependence currently on direct code in some tactical system software. However, it reveals that the weightings in the application vector (paragraph 2c above) represent compromises between a variety of intended applications, since the area in question (that of a standard language for tactical software) is fairly broad. In one of the particular subareas concerned with tactical applications, the weightings for power and efficiency would be somewhat higher, and those for other goals appropriately lower, than the actual values in the Application Vector. For such a subarea, machine language insertions would receive a higher score in the Intermediate Vector.

b. The sum of the entries in $IV^U$ is 541.4; thus the transformation constant (see paragraph I.4.g in this chapter) is K(U) = .185.

TABLE II. PRODUCT OF RATING MATRIX AND APPLICATION VECTOR

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 7.47 | 7.90 | 6.75 | 5.32 | 5.57 | 7.85 | 7.35 | | | | |
| B | 7.62 | 7.47 | 7.32 | 7.27 | 5.85 | 7.57 | 5.15 | 7.82 | 6.32 | 7.45 | 6.32 |
| C | 7.30 | 6.77 | 7.35 | 6.95 | 7.07 | 7.52 | 7.77 | 6.77 | 6.77 | | |
| D | 7.32 | 7.72 | 7.07 | 7.27 | 7.60 | 5.80 | | | | | |
| E | 7.42 | 7.92 | 7.20 | 5.72 | 8.35 | 8.35 | 7.20 | 8.02 | | | |
| F | 6.97 | 8.00 | 7.72 | 8.07 | 7.77 | 7.67 | 7.60 | | | | |
| G | 7.80 | 6.97 | 7.87 | 7.87 | 6.12 | 7.10 | 7.30 | 7.30 | | | |
| H | 7.20 | 7.05 | 5.67 | 6.38 | 6.45 | 6.65 | 5.55 | 6.65 | 6.38 | 6.65 | |
| I | 6.90 | 6.62 | 6.50 | 6.97 | 7.07 | 5.35 | 5.00 | | | | |
| J | 5.82 | 6.97 | 4.00 | 5.85 | 5.80 | | | | | | |

# Section III. LANGUAGE-INDEPENDENT ASPECTS OF THE OVERALL EVALUATION

## 1. Background.

a. High Cost of Software. The basic issue which underlies this study is that of the high costs of software, and, in particular, the role of the language with respect to these costs. There are two factors involved with high costs which must be considered. First, part of the expense of software can be attributed to the expansion in the applications for which computers are being used. Second, part of the costs are indirect, due to inefficiencies in the production of the software which are avoidable with the proper managerial/technical facilities. The choice of proper language is relevant to both factors. With respect to the first area, the language must contain features that enable the necessary applications to be programmed at all. Concerning the second, in view of the fact that program verification and maintenance have been identified as the software development stages demanding the most resources -- particularly, programmer time -- the language (or the tools supporting its use) must promote the use of sound programming methodology.

b. "Non-Technical" Factors in Choosing a Language. The evaluation tool used in this report accounts for these factors associated with high software costs, primarily through the technical evaluation stage. Clearly, however, there are other criteria which enter into a language selection decision, having to do mainly with the existence of the language, and the history of its use (e.g., compiler and support tool availability). Unfortunately, these other criteria do not always correlate directly with the technical merit of the language. In fact, the older the language, the more likely it is for the language to have wide usage and good support tools, but the less likely for it to fare well against technical standards oriented around state-of-the-art research.

c. The Importance of a HOL's Technical Merit. The decision as to whether the benefits of an existing language outweigh the advantages of a technically superior but unimplemented language[1] must be based on the individual

[1] In the term "unimplemented language", we include both a language which has no implementation, and a language which has an existing implementation, but which has been modified in its design for technical improvement and for which these modifications have not been implemented.

circumstances surrounding the language evaluator's computing environment. For a private industry which is constrained by a particular set of computing equipment, the costs associated with implementing a technicably superior but unimplemented language would probably outweigh the benefits, and the use of an existing language might be more cost-effective. However, when one considers software procurement and development agencies such as the Army, or DoD, then technical merit tends to be the decisive criterion, based on the relatively small costs involved with language implementation compared with expected savings from the use of a better language. With DoD software costs in the range of $3 billion annually, the potential savings for each percent come to $30 million per year. Just a one-percent savings from the use of a technically superior language would more than offset any needed development and implementation costs.

d. Approach to Defining the Management Decision-Making Criteria. For the purposes of the Overall Evaluation stage of the evaluation tool, a set of management decision-making criteria is described in the next paragraph. These criteria take into account technical factors as well as considerations relating to existing implementations, including the relevant requirements from the "*Tinman's Needed Characteristics*," Sections K, L, and M. The discussions of the various criteria define procedures which may be carried out to establish quantitative scores for any candidate HOL's Language Management vector. However, because the effort required to perform these procedures for the non-technical factors was beyond the scope of the contract, this report does not present numeric scores for the candidate HOLs. Instead, qualitative assessments are given.

2. Management Decision-Making Criteria.

Paragraphs a through e below describe procedures for obtaining quantitative estimates, in the range 0 to 100, for any candidate HOL's degree of compliance with the management decision-making criteria.

a. Technical Merit. The technical merit of the language is the result of the technical evaluation stage. It is the only criterion which is independent of the existence of an implementation for the language.

b. Availability. The score for a language's availability is derived from the results of a questionnaire (Appendix I) distributed within the Army. It is intended that copies of this questionnaire be sent to each programming group

manager conducting an in-house project, and to each technical monitor who oversees a contractor programming effort.

(1) The questionnaire consists of two parts, and a separate questionnaire is to be filled out for each language in use by the members of the group. The first part requests general information on the reasons for using the language, and the kinds of hardware on which the language is implemented. This information, though not directly impacting the language's score for the availability criterion, could be used subsequently in an analysis of the reasons for the presence of those languages which are available. The first part of the questionnaire also requests that the respondent specify the total number of programmers using the language. This number will then be used as a weighting factor in the determination of the availability score. The way this is done is as follows. Assume that there are r respondents and that a total of c candidate languages are mentioned in the responses. We denote by P(i,j) the number of programmers in the site of the $i^{th}$ respondent who use the $j^{th}$ language. (The questionnaire explains how to avoid duplicating a count when the same programmer uses several languages parttime.) P(*,j) is the total number of programmers using language j -- i.e., $P(*,j)=\sum_{i=1}^{r} P(i,j)$.

The weighting factor W(j) for the $j^{th}$ language is defined to be P(*,j) / P(*,jmax) where jmax is the index of the language with the largest total number of programmers: i.e., $P(*,j) \leq P(*,jmax)$ for j=1,2,...,c. The use of this weighting factor is explained below.

(2) The second part of the questionnaire requests scores and weightings relevant to the availability of the language. This part comprises four sections: compiler, tools, documentation, and other. The respondent, by supplying the information requested, will induce separate scores (between 0 and 10) for the availability of the compilers, tools, documentation, and any other values he wishes to include. He is asked to provide weightings (values between 0.0 and 1.0 which sum to 1.0) for these four components

which reflect the relative importance of these components for his particular programming group. The result of summing the weighted scores is a score for the availability of language j at site i; we denote this score by S(i,j).

(3) The score for the availability of the $j^{th}$ language, over all sites, is then calculated using the derived weighting factors W(j) and availability scores S(i,j): this result is

$$W(j) * \frac{10}{r} * \sum_{i=1}^{r} S(i,j),$$

and will be a value between 0 and 100. Note that, for the most available language (the $jmax^{th}$), the weighting factor is simply 1.

(4) It should be realized that one of the most critical aspects of this process, in terms of the impact on the final score for availability, is how to weight the responses. (The weighting question arises also in the derivation of a single score for each respondent, but we solved this problem by asking the respondent himself to supply the weightings for the various constituents according to the requirements and priorities of his project.) For example, one approach is to consider all respondents equal -- but this fails to distinguish availability for large vs. small projects. Our method here is to account for this by using as the weighting factor the percentage of programmers using the language. But even here there are several possible techniques. For example, the weighting could be a pure percentage -- if 50% use language A, 30% use language B, and 20% use language C, the weights would be .5, .3, and .2, respectively. This has the effect of partitioning the maximum availability score into components: i.e., language A could then score no more than 50, B could do no better than 30, and C is constrained to be less than or equal to 20. This was not the approach used with the score for technical merit, however; in the latter, each language was judged independently against a single model. We have attempted to use a similar strategy here; the single model in this case is the language with

maximum availability. Thus the weighting factor for this language is automatically 1, and the others are derived relative to this standard. In the above example, the weights for languages A, B, and C would be 1 (.5/.5), .6(.3/.5), and .4(.2/.5).

c. Standardization Status. One of the objectives of adopting a common HOL is to eliminate or prevent incompatible dialects. To derive a score for this criterion, we use the following factors:

(1) The existence of a complete, unambiguous definition, with a minimum of implementation dependencies, which specifies not only the legal programs in the languages but the actions to be taken when a program is illegal.

(2) The existence of a formalized procedure (e.g., a set of test programs -- both legal and illegal) to be used for validating a compiler vs. the language definition.

(3) The existence of a well-defined, orderly procedure for permitting the evolution of the language based on user feedback and advances in the state-of-the-art in language design and programming methodology.

(4) The existence of a committee, with DoD involvement, embodied with responsibilities for maintaining the language definition standard and answering questions from implementors.

(5) The existence of a journal (e.g., COBOL's Journal of Development) devoted specifically to the language.

Each factor will be rated, from 0 to 10, based on the standardization status of the language being evaluated. These scores will be weighted as follows: .4 for (1), .15 for (2), .15 for (3), .2 for (4), and .1 for (5). The following weighted sum (between 0 and 100) will be used as the value for standardization status:

$$10 * \sum_{i=1}^{5} \text{Weight}(i) * \text{Score}(i)$$

d. Industry Support.

(1) This is a difficult factor to quantify. Ideally, a questionnaire of the sort used to determine availability (Appendix I) would be useful, but it is extremely doubtful that a company would voluntarily offer such information, due to the possibly privileged nature of the data, and to the time required to complete the questionnaire. The alternative suggested here is to attempt to obtain information concerning at least the production of compilers for the various languages under consideration. This can be done by sending a brief questionnaire to the 100 largest companies in the data processing industry. This questionnaire will request that the company (i) list each compiler and language-specific software aid currently available, and (ii) specify the hardware environment needed for each item in (i).

(2) The reduction of the data obtained from responses to these questionnaires, to derive a single score between 0 and 100, proceeds as follows. First, to give compilers more weight than support tools, we assign three points to each compiler and one point to each support tool. For any given response, such a point count is multiplied by the number of different machines for which the compiler or tool is implemented. Let $P(i,j)$ be the total number of points derived from the $i^{th}$ respondent with respect to the $j^{th}$ language;

$$P(*,j)=\sum_{i=1}^{100} P(i,j)$$

is the total number of points for the $j^{th}$ language.

(3) The basic idea is to measure industry support relative to some standard. As in the calculation of availability, we here use as the standard the language most supported (as revealed in the responses). This is done in two ways (one which treats all companies as equal, the other which weighs larger companies more heavily); the average of these two is then used as the score for industry support. According to the first method, we define jmax as the index of the language with

the highest total points; i.e., $P(*,jmax) \geq P(*,j)$ for all j. Then the score for industry support for language j, denoted $IS_E(j)$ -- The E subscript denotes the fact that all companies are considered equal -- is simply $100*P(*,j)/P(*,jmax)$. The second method determines the score for industry support of language j, taking weighting into account; this score is denoted $IS_W(j)$. To calculate the weights, assume that the total data processing revenues for the $i^{th}$ company are D(i) dollars (this information is obtainable from various sources, such as [11]). Let

$$D(*) = \sum_{i=1}^{100} D(i)$$

and define the weight W(i) to be D(i)/D(*). Let the weighted point total for company i and language j, denoted WP(i,j), be the product W(i)*P(i,j). Define WP(*,j) to be

$\sum_{i=1}^{100} WP(i,j)$, the weighted point total for language j. Analogous to the first method, let jmaxw be the index of the language with the largest weighted point total - i.e., $WP(*,jmaxw) \geq WP(*,j)$ for all j. Finally, the score $IS_W(j)$ is defined to be $100*WP(*,j)/WP(*,jmaxw)$. And the score for the industry support of language j, or IS(j), is simply $(IS_E(j) + IS_W(j))/2$.

(4) As an example of this calculation, assume that Company 1's response results in 5 points for Language A, 3 points for Language B, and 0 points for Language C; for Company 2, we have 1 point for A, 2 for B, 5 for C; for Company 3, we have 2 points for A, 5 for B, and 1 for C. Thus, $P(*,A) = 8$, $P(*,B) = 10$, and $P(*,C) = 6$; jmax = B; and $IS_E(A) = 100*8/10$, $IS_E(B) = 100*10/10$, and $IS_E(C) = 100*6/10$. Assume that Company 1's percentage of total revenue is 50%; for Company 2, it is 30%; and for Company 3, it is 20%. The weighted point totals are as follows: for Company 1, A has 2.5 points and B has 1.5; for Company 2, A has 0.3 points, B has 0.6, and C has

1.5; for Company 3, A has 0.4 points, B has 1.0, and C has 0.2 points. Now, WP(*,A) = 3.2, WP(*,B) = 3.1, and WP(*,C) = 1.7; jmaxw = A; and $IS_W(A) = 100*3.2/3.2$, $IS_W(B) = 100*3.1/3.2$, $IS_W(C) = 100*1.7/3.2$. Thus, IS(A) = 90, IS(B) = 98, and IS(C) = 57.

e. Availability of Trained Programmers. A score for the availability of programmers for a language can be obtained using questionnaires distributed to programming group managers within the Army. Each respondent would be requested to supply the following information for each language used in his project:

(1) How many programmers are currently available to use the language on current (or planned) activities?

(2) How many programmers for the language are required to perform the current (or planned) activities?

Let PA(i,j) be the numbers of Programmers Available at the $i^{th}$ site for the $j^{th}$ language, and let PR(i,j) be the number of Programmers Required. Define the total number of programmers available for the $j^{th}$ language, PA(*,j), as $\Sigma_i PA(i,j)$; similarly, PR(*,j) = $\Sigma_i PR(i,j)$. The score for the availability of the $j^{th}$ language is now defined as 100 if PA(*,j) $\geq$ PR(*,j), or 100*PA(*,j)/PR(*,j), otherwise.

3. Effects of Management Decision-Making Criteria on Cost.

a. Scope. As described in the outline of the Overall Evaluation (paragraph I.5d of this chapter), we intend for the scores in the Language Management vector to be used in the projection of estimates for the various cost categories. A fundamental step in the process is the derivation of the functions that relate the Language Management elements to the costs. However, the effort required to carry out this derivation was beyond the scope of the contract. As a result, the relationships between the management decision-making criteria and the software cost categories are described below in a qualitative manner.

b. Language Acquisition. The costs associated with acquiring a language processor, support tools, and documentation, are obviously strongly dependent on the availability of these elements. If they are not available in the computing environment in which they will be used, then the costs are influenced directly by industry support and standardization status, since it may be possible to obtain the language commercially. The technical merit affects acquisition costs if the compiler and tools must be implemented "from scratch" or transported from a different computing environment. In the latter case, the "maintainability" and "transportability" components of the technical merit will be most directly relevant. Programmer availability has minor impact on acquisition costs (programmer availability pertains to the language being acquired).

c. Programmer Training. Programmer availability is the factor that most affects training costs. Training presupposes the existence of a language processor, support tools, and documentation; the costs for acquiring these elements were taken into account in paragraph b above. Standardization status and industry support affect programmer training costs only indirectly, through their impact on language acquisition. A language's technical merit can have a substantial influence on programmer training costs; the "learnability" and "power" components of the technical merit are the scores most directly relevant.

d. System Development. The implementation of a software system requires a language processor environment and trained programmers; the factors affecting these costs were considered in paragraphs b and c above. Language and programmer availability, standardization status, and industry support influence system development costs indirectly, through their influence on language acquisition and programmer training. The major factor that directly affects system development costs is the language's technical merit; as described in paragraph II.2a of this chapter, each component of the technical merit is applicable at some stage of the system development process. Of special relevance are the "reliability" and "maintainability" components of technical merit, in view of the relatively high percentage of software life-cycle costs that are devoted to debugging and maintenance.

e. Indirect Expenses. The indirect expenses (i.e., the costs resulting from software that does not meet its performance requirements) are by their nature the most difficult to estimate. Relationships between the management decision-making criteria and these expenses are system-specific; in fact, these expenses are influenced more by the way in which the system development is managed than by the management decision-making criteria. As a result, a study of the subject of indirect expenses proved to be outside the scope of this report.

4. Directions for Further Work.

There are two aspects of the Overall Evaluation that require further study before the techniques proposed become fully practicable. First is the derivation of quantitative measures for the various management decision-making criteria; this could rely on paragraph II.2 as its basis. Second, and more difficult, is the establishment of functions that relate the scores for the decision-making criteria to the actual costs involved with system implementation. A possible approach here is to use the actual costs associated with the implementation of existing software systems, so that a relationship between the decision-making criteria and these costs could be fitted.

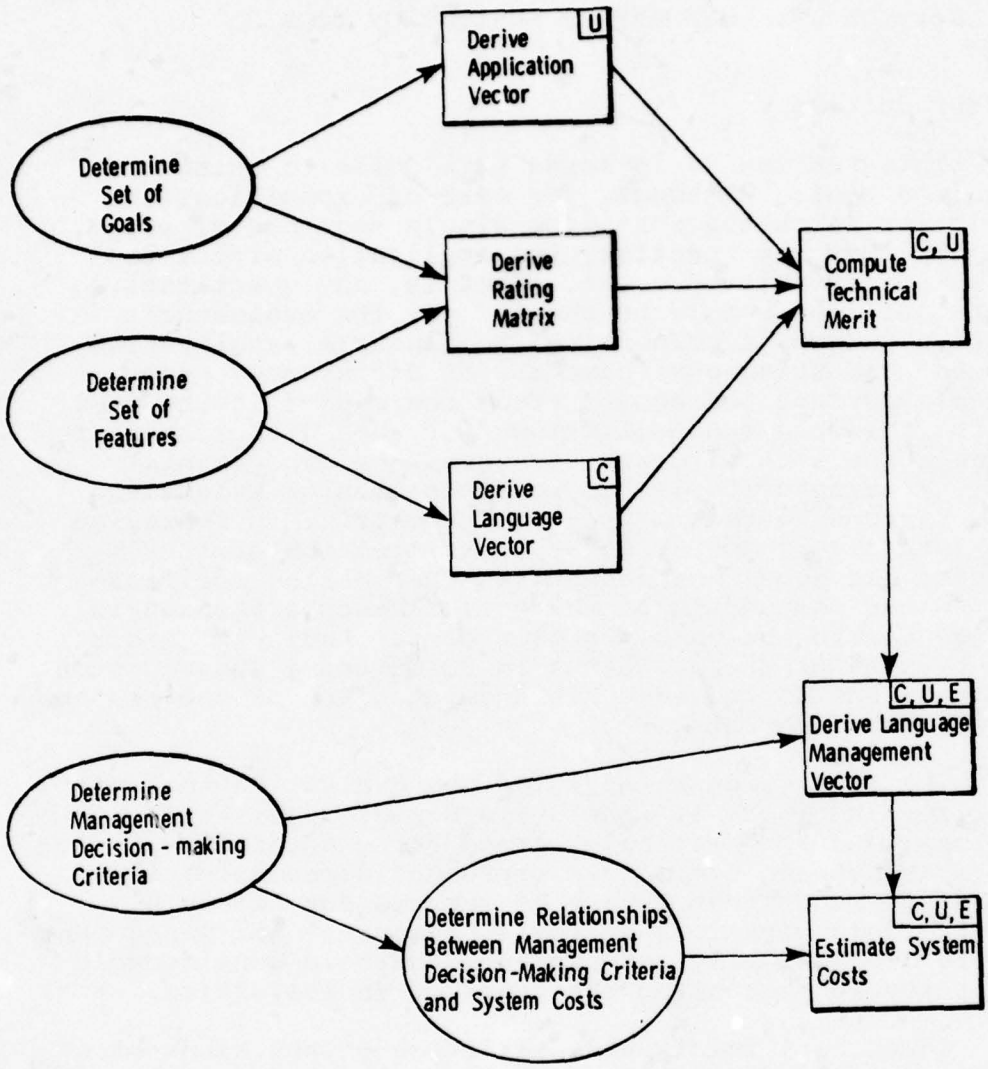
## Section IV. SUMMARY OF EVALUATION TOOL

1. Introduction.

a. This section is intended as a guide to using the evaluation tool. Although, for ease of exposition, the description takes the form of a single sequence of steps, we expect that, in practice, the application of the tool will be an iterative process. That is, any quantitative results obtained should be checked vs. the evaluator's intuitive judgments concerning the language, application area, and computing environment. If differences occur, the evaluator can and should trace the causes in the tool itself. Possible reconciliations for such differences are to change the sets of goals, features, or management decision-making criteria; to revise scores or weightings in the various vectors or the rating matrix; or to revise one's intuitive judgment after re-examination of the evidence. It should be emphasized that making modifications to the components of the tool in such a fashion is consistent with the main purposes of the tool - to assist (not to replace) the evaluator in formulating judgments on a language and to increase his understanding of the reasons behind those judgments.

b. The main steps in applying the evaluation tool are summarized in Figure 1. The ovals depict qualitative decisions, and the rectangles stand for quantitative ones. The arrows between components represent dependence; "[A] → [B]" means that A must be carried out before B. Since the determinations of goals, features, and management criteria are logically independent, there is considerable flexibility in sequencing the steps of an evaluation.

c. There is a fairly wide variation in the kinds of activities implicit in the several stages of the evaluation tool's application. The component of Figure 1 marked "Compute Technical Merit" can be carried out mechanically, given the necessary matrix and vectors. A summary of these steps is given in Table III. On the other hand, the determination of goals, features, and criteria, and the derivation of scores and weightings for the rating matrix and various vectors, are not mechanizable, and a technique such as the Delphi method (supplemented by a questionnaire analysis as described in paragraph 2.III.2) may be appropriate if resources are available. (This last qualification is relevant, since the number of goals, features, and criteria which are needed to ensure completeness may result in extremely large vectors or matrices.




KEY: " C " indicates dependence on candidate language C
" U " indicates dependence on application area U
" E " indicates dependence on computing environment E

Figure 1. Stages in the Application of the Evaluation Tool

# TABLE III. TECHNICAL EVALUATION

| | R * | $A^U$ → | $IV^U$ |
|---|---|---|---|
| NAME | Rating Matrix | Application Vector | Intermediate Vector |
| SIZE | m x n<br>m = # of features<br>n = # of goals | n x 1 | m x 1 |
| INFORMATION CONTENT | $R_{ij}$ = contribution of $i^{th}$ feature toward $j^{th}$ goal<br>Language-independent<br>Application-independent | $A_j^U$ = weighting of $j^{th}$ goal with respect to application area U<br>Language-independent<br>Application-dependent | $IV_i^U$ = contribution of $i^{th}$ feature to application area U<br>Language-independent<br>Application-dependent |
| ELEMENT SIZE | $0 \leq R_{ij} \leq 10$ | $0.0 \leq A_j^U \leq 1.0$<br>$\sum_{j=1}^{n} A_j^U = 1.0$ | $0 \leq IV_i^U \leq 10$<br>$IV_*^U$ = sum of entries in $IV^U$ |
| REFERENCES (in Ch. 2) | Paragraph I.4a, II.3 (Table I) | Paragraph I.4e, II.c | Paragraph I.4f, II.4 (Table II) |

| | K(U) * | $(L^C$ * | $IV^U)$ → | t |
|---|---|---|---|---|
| NAME | | Language Vector | Intermediate Vector | Technical Merit |
| SIZE | Scalar | 1 x m<br>m = # of features | m x 1 | Scalar |
| INFORMATION CONTENT | Transformation constant<br>Language-independent<br>Application-dependent | $L_i^C$ = degree to which $i^{th}$ feature is present in language C<br>Language-dependent<br>Application-independent | $IV_i^U$ = contribution of $i^{th}$ feature to application area U<br>Language-independent<br>Application-dependent | Technical merit of language C in application area U<br>Language-dependent<br>Application dependent |
| ELEMENT SIZE | $100/IV_*^U$ | $0 \leq L_i^C \leq 1.0$ | $0 \leq IV_i^U \leq 10$ | $0 \leq t \leq 100$ |
| REFERENCES (in Ch. 2) | Paragraph I.4g | Paragraph V.1,...,X.1 (Tables IV through IX) | Paragraph I.4f, II.4 (Table II) | Paragraph V.1,...,X.1 |

For example, the number of elements in the Rating matrix could easily be in the thousands.)

2. Summary of Technical Evaluation.

a. Determining Features and Goals. A preliminary step is the identification of (1) a spanning set of features found in HOLs, and (2) a complete set of goals that are relevant to HOL software.

b. Derivation of Rating Matrix. Once the features and goals have been determined, the derivation of entries in the Rating matrix R may proceed. The score $R_{ij}$ is to reflect the degree to which the $i^{th}$ feature contributes toward the $j^{th}$ goal; this value is to be between 0 and 10, inclusive.

c. Derivation of Application Vector. Once the set of goals is determined, the Application vector $A^U$ may be derived for application area U. The score $A^U_j$ is the weighting of the $j^{th}$ goal with respect to application area U. The scores in the application vector must be non-negative and sum to 1.0.

d. Derivation of Language Vector. After the set of features has been determined, the Language vector $L^C$ may be derived for candidate language C. The score $L^C_i$ is to reflect the degree to which the $i^{th}$ feature is supported by language C; this value is to be between 0.0 and 1.0, inclusive.

e. Computation of Intermediate Vector. From the Rating matrix R and the Application vector $A^U$, the Intermediate vector $IV^U = R*A^U$ is computable. The score $IV^U_i$ reflects the degree to which the $i^{th}$ feature contributes toward application area U, and is in the range of 0 to +10.

f. Computation of Transformation Constant. The scalar value $L^C*IV^U$ is between 0 and $IV^U_*$, inclusive, where $IV^U_*$ is the sum of entries in $IV^U$. The constant K(U) is defined as $K(U) = 100/IV^U_*$.

g. Computation of Technical Merit. The scalar value $K(U)*(L^{C}*IV^{U})$ is a result in the range 0 to 100 that reflects the technical merit of language C with respect to application area U.

3. Summary of Overall Evaluation.

a. Determining Management Decision-Making Criteria. The first step of the overall evaluation is the establishment of a set of criteria that will be used to govern the selection of a HOL. It is assumed that technical merit will be one of these criteria.

b. Derivation of Language Management Vector. After the set of management criteria is determined, the Language Management vector $LM^{C,U,E}$ may be derived for language C with respect to application area U and computing environment E. The score $LM^{C,U,E}$ is to reflect the merit of language C with respect to the $k^{th}$ criterion, assuming that the HOL is used for application area U in computing environment E. Each such value is in the range 0 to 100.

c. Determining Relationships between Management Decision-Making Criteria and System Costs. After the set of management decision-making criteria is established, the mathematical functions that relate the scores for these criteria to actual system costs may be determined. This is likely to be the most difficult step to carry out.

d. Estimation of System Costs. Using the Language Management vector derived in b in conjunction with the relationships established in c yields an estimated value for the cost of implementing the given software system in candidate language C. A comparison of these estimates for the candidate HOLs will then reveal the cost-effectiveness of the various choices.

## Section V. TACPOL AND THE MANAGEMENT DECISION-MAKING CRITERIA

This section presents the score for TACPOL's technical merit and describes informally the language's compliance with the "non-technical" criteria pertinent to HOL selection.

### 1. Technical Merit.

The language vector for TACPOL is displayed in Table IV. Multiplying the vector by the product of the Rating Matrix and the Application vector (see paragraph II.4) results in the scalar value 264.9 with respect to the possible range 0 through 541.4; the corresponding value in the range 0 to 100 (after being rounded to two significant figures) is 49.

TABLE IV. LANGUAGE VECTOR FOR TACPOL

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A | .5 | .6 | .0 | .7 | .2 | .6 | .5 | | | | |
| B | .4 | .4 | .7 | .9 | .0 | .7 | .0 | .5 | .4 | .5 | .5 |
| C | .0 | .9 | 1.0 | .0 | .6 | .3 | .6 | .0 | .0 | | |
| D | .6 | .9 | .0 | .8 | .6 | .0 | | | | | |
| E | .0 | .0 | .9 | .0 | .0 | .6 | .5 | .0 | | | |
| F | .4 | .4 | 1.0 | .8 | .8 | .8 | .7 | | | | |
| G | .7 | .8 | .8 | .9 | .1 | .0 | .1 | .1 | | | |
| H | 1.0 | 1.0 | 1.0 | .9 | .5 | .3 | .8 | 1.0 | .8 | .7 | |
| I | .2 | .4 | .0 | .0 | .8 | .5 | 1.0 | | | | |
| J | .9 | .0 | .8 | .6 | .0 | | | | | | |

2. Availability.

a. Compilers. Despite the fact that TACPOL's simplicity and orientation toward the AN/GYK-12 machine characteristics should permit the construction of good compilers, this area has been a source of problems for users of the language. Difficulties such as the generation of inefficient or incorrect code have resulted in heavier use of MOL in TACFIRE and TOS than would otherwise have been necessary. There has also been a minor but persistent problem with character encodings (specifically, conflicts with IBM's ASCII representations for "not" and "or").

b. Tools. There is room for improvement in the support tools provided for TACPOL. Desirable additions include better formatting and cross-referencing for output listings, a macro facility and a provision for inserting source code in a TACPOL program at compile-time, more extensive COMPOOL and library facilities, and better maintenance and diagnostic tools.

c. Documentation. The TACPOL documentation [3 and 6] is adequate, with [3] providing a formal definition and [6] serving more as a user's guide. The execution model presented in [3] is particularly helpful in describing the behavior of the various language constructs.

3. Standardization Status.

TACPOL does not satisfy very well the criterion of possessing a strongly enforced, formal standardization program; however, it has never been an objective to produce implementations of the language on a variety of machines.

4. Industry Support.

There is no support or use of the TACPOL language outside of the Army and Litton and other contractors working on Army tactical systems.

5. Availability of Trained Programmers.

Because of the simplicity of the language, training costs for TACPOL programmers should be relatively low; currently, a TACFIRE programming course at USACSC includes about a week and a half on TACPOL.

6. Effects of Criteria on Costs.

a. Language Acquisition. Costs in this area are relatively small, in view of the availability of the language in the Army. Compiler and support tool improvement are areas in which future expenses are likely.

b. Programmer Training. As mentioned in paragraph 5 above, this factor can be expected to be the smallest component of language-related cost.

c. System Development. The technical inadequacies in TACPOL and the failings with respect to compiler performance and support tools contribute heavily toward high costs in this area. The use of a technically superior language, together with a more complete and reliable language support system, could effect noticeable reductions of system development costs.

## Section VI. CS-4 AND THE MANAGEMENT DECISION-MAKING CRITERIA

This section presents the score for CS-4's technical merit and describes informally the language's compliance with the "non-technical" criteria pertinent to HOL selection.

### 1. Technical Merit.

The language vector for CS-4 is displayed in Table V. Multiplying this vector by the product of the Rating Matrix and the Application vector (see paragraph II.4) results in the scalar value 410.2 with respect to the possible range 0 through 541.4; the corresponding value in the range 0 to 100 (after being rounded to two significant figures) is 76.

TABLE V. LANGUAGE VECTOR FOR CS-4

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 1.0 | .9 | .4 | .2 | .0 | .9 | 1.0 | | | | |
| B | .9 | .9 | .7 | 1.0 | 1.0 | 1.0 | .4 | .9 | 1.0 | .9 | .9 |
| C | 1.0 | .9 | 1.0 | .8 | .2 | .9 | .6 | .7 | .0 | | |
| D | 1.0 | .9 | .8 | 1.0 | .9 | .0 | | | | | |
| E | .8 | .3 | 1.0 | .0 | 1.0 | 1.0 | .8 | 1.0 | | | |
| F | 1.0 | 1.0 | 1.0 | .7 | 1.0 | 1.0 | .9 | | | | |
| G | .9 | .9 | .8 | .8 | .0 | .9 | .8 | .6 | | | |
| H | .8 | 1.0 | .4 | .9 | .8 | .8 | .7 | 1.0 | .1 | 1.0 | |
| I | .4 | .6 | .3 | .2 | .3 | .2 | 1.0 | | | | |
| J | .8 | 1.0 | .9 | .8 | 1.0 | | | | | | |

2. Availability.

Since CS-4 has not been implemented, there are no compilers or support tools available. The reference manual and operating system interface description [4 and 5] are adequate for informal documentation, but a more complete and formal specification is needed.

3. Standardization Status.

There is no formal standardization program for CS-4.

4. Industry Support.

As an unimplemented language, CS-4 has no industry support.

5. Availability of Trained Programmers.

Again, because of the absence of implementations, CS-4 fails to meet this criterion.

6. Effects of Criteria on Costs.

a. Language Acquisition. The lack of implementations makes this cost factor significant. The effort required would be to polish the design, to implement the language and a set of support tools, and to provide user documentation.

b. Programmer Training. Once the language is implemented, expenses in this area should not be large.

c. System Development. The technical superiorities of CS-4 over the other five candidate HOLs suggest that the costs for system development will be relatively low if CS-4 is used.

## Section VII. JOVIAL AND THE MANAGEMENT DECISION-MAKING CRITERIA

This section presents the score for JOVIAL's technical merit and describes informally the language's compliance with the "non-technical" criteria pertinent to HOL selection.

### 1. Technical Merit.

The language vector for JOVIAL is displayed in Table VI. Multiplying this vector by the product of the Rating Matrix and the Application vector (see paragraph II.4) results in the scalar value 274.0 with respect to the possible range 0 through 541.4; the corresponding value in the range 0 to 100 (after being rounded to two significant figures) is 51.

TABLE VI. LANGUAGE VECTOR FOR JOVIAL (J73/I)

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A | .5 | .7 | .4 | .1 | .0 | .7 | .4 | | | | |
| B | .6 | .6 | .9 | .9 | .6 | .8 | .0 | .3 | .3 | .0 | .7 |
| C | .0 | .8 | 1.0 | .6 | .6 | .4 | .6 | .1 | .0 | | |
| D | .3 | .6 | .4 | .2 | .7 | .2 | | | | | |
| E | .4 | .0 | 1.0 | .0 | .0 | .6 | .5 | .0 | | | |
| F | .8 | .6 | 1.0 | .8 | 1.0 | 1.0 | .0 | | | | |
| G | .6 | .6 | .8 | .6 | .1 | .0 | .1 | .0 | | | |
| H | .7 | 1.0 | 1.0 | .7 | .5 | .7 | .8 | 1.0 | .0 | .6 | |
| I | .1 | .6 | 1.0 | .9 | .8 | .5 | .8 | | | | |
| J | .9 | .0 | .3 | .7 | .0 | | | | | | |

2. Availability.

JOVIAL (specifically, J73/I) is quite weak in this area. There are currently only two implemented compilers, and the language does not enjoy a very heavy usage. In fact, there is no usage of JOVIAL outside the Air Force. With respect to documentation, the reference manual [14] could be improved. The organization of this document was not clear, the BNF rules were presented neither top-down nor bottom-up, and the explanations in the semantics sections were not always complete.

3. Standardization Status.

Although JOVIAL might be expected to rate highly with respect to this criterion, in actual experience the standardization program has not prevented the development of numerous incompatible dialects (of previous versions of JOVIAL). The absence of I/O from [14] suggests that similar problems may occur in the future.

4. Industry Support.

One of the main failures of the JOVIAL effort in the past has been in precisely this area; there has never been a substantial amount of support (i.e., compiler and tool development) in industry.

5. Availability of Trained Programmers.

The number of JOVIAL programmers available to the Army is negligible, because of both the newness of the language and the unavailability of JOVIAL outside the Air Force.

6. Effects of Criteria on Costs.

a. Language Acquisition. The costs here involve the production of new code generators for existing JOVIAL compilers, to make the language available on Army hardware, and the development of language support tools.

b. Programmer Training. This expense should be relatively small, since JOVIAL is not a large language.

c. System Development. Costs in this area will likely be roughly comparable to those resulting from TACPOL usage. Reductions could be effected with the use of a technically superior HOL.

## Section VIII. FORTRAN AND THE MANAGEMENT DECISION-MAKING CRITERIA

This section presents the score for FORTRAN's technical merit and describes informally the language's compliance with the "non-technical" criteria pertinent to HOL selection.

### 1. Technical Merit.

The language vector for FORTRAN is displayed in Table VII. Multiplying this vector by the product of the Rating Matrix and the Application vector (see paragraph II.4) results in the scalar value 222.7 with respect to the possible range 0 through 541.4; the corresponding value in the range 0 to 100 (after being rounded to two significant figures) is 42.

TABLE VII. LANGUAGE VECTOR FOR FORTRAN

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A | .4 | .7 | .1 | .2 | .1 | .7 | .1 | | | | |
| B | .3 | .4 | .7 | .9 | .8 | .7 | .0 | .5 | .3 | .5 | .5 |
| C | .0 | .9 | 1.0 | .7 | 1.0 | .5 | .7 | .0 | .0 | | |
| D | .9 | .7 | .7 | .0 | .3 | .0 | | | | | |
| E | .1 | .0 | .8 | .0 | .0 | .2 | .5 | .0 | | | |
| F | .6 | .3 | .8 | 1.0 | .0 | .0 | .5 | | | | |
| G | .4 | .6 | .4 | .4 | .0 | .0 | .3 | .0 | | | |
| H | .5 | 1.0 | 1.0 | .9 | .0 | .5 | .7 | 1.0 | .9 | 1.0 | |
| I | .0 | .0 | .0 | .0 | .8 | .5 | 1.0 | | | | |
| J | .9 | .2 | .0 | .0 | .0 | | | | | | |

2. Compiler, Support Tool, and Documentation Availability.

a. The widespread availability of FORTRAN has been one of the greatest advantages that FORTRAN has held over its competitors. The set of support tools developed is quite broad, ranging from software instrumentation packages to language preprocessors that enable users to code in more "structured" dialects. The documentation for the language has also been a strong point for FORTRAN; there are numerous textbooks that either teach the language or use it for specifying algorithms.

b. It should be pointed out, however, that the draft proposed FORTRAN [1] used for evaluation in this report does not possess any of these advantages at present, in view of the large differences between [1] and the previous standard.

3. Standardization Status.

FORTRAN has had a USANSI standard since 1966, but there are nearly as many different FORTRAN versions as there are compiler writers. Each version has its own set of extensions and restrictions. The result has been that, except for straightforward numerical programs doing no character processing or I/O, FORTRAN programs have not generally been portable. With the acceptance of the proposed draft standard [1], this situation should improve, because many operations that could be done only with "tricks" in earlier versions have become part of the language, and other operations have become better defined. Implementation-dependencies remain in the definition of floating-point precision, so that a calculation requiring only single precision on one machine may require double precision on another.

4. Industry Support.

The situation with respect to industry support is similar to that described above concerning availability: the experience with previous FORTRAN standards has been strong support throughout the computer industry, but the version of the language proposed in [1] is too recent to base a judgment upon.

5. Availability of Trained Programmers.

FORTRAN's widespread use in the technical and scientific communities and as a teaching language in

universities has resulted in a large supply of trained programmers, and this situation is also true in the Army. Again, however, this pertains to previous versions of FORTRAN, not [1]. The similarity between the languages facilitates retraining existing programmers, and the improvements introduced should lighten the task of teaching new programmers. However, it should also be noted that the teaching of FORTRAN in academic environments appears to be declining, as many universities are emphasizing languages that are somewhat better suited to the description of complex algorithms.

6. Effects of Criteria on Costs.

a. Language Acquisition. Because of the large differences between the draft proposed and the previous FORTRAN standard, acquiring the language defined by [1] entails implementation of new compilers and probably new support tools, as well as the production of new documentation. The costs involved would not be so large as the analogous expenses for CS-4, but they would still be significant.

b. Programmer Training. This expense should not be sizable.

c. System Development. Since FORTRAN is not as sound technically as other candidate HOLs considered in this report, system development costs are likely to be higher than for the other languages. We also note here that the absence of direct code disqualifies FORTRAN from consideration (unless the language is modified), since such a facility is an absolute requirement for the area of embedded computer applications.

## Section IX. COBOL AND THE MANAGEMENT DECISION-MAKING CRITERIA

This section presents the score for COBOL's technical merit and describes informally the language's compliance with the "non-technical" criteria pertinent to HOL selection.

### 1. Technical Merit.

The language vector for COBOL is displayed in Table VIII. Multiplying this vector by the product of the Rating Matrix and the Application vector (see paragraph II.4) results in the scalar value 252.7 with respect to the possible range 0 through 541.4; the corresponding value in the range 0 to 100 (after being rounded to two significant figures) is 47.

TABLE VIII. LANGUAGE VECTOR FOR COBOL

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A | .4 | .4 | .0 | 1.0 | .5 | .6 | .7 | | | | |
| B | .7 | .6 | .8 | .9 | .5 | .6 | .1 | .0 | .9 | .9 | .0 |
| C | .7 | .7 | .0 | .0 | 1.0 | .0 | .2 | .0 | .0 | | |
| D | .2 | 1.0 | .4 | .7 | .6 | .0 | | | | | |
| E | .0 | .0 | 1.0 | .0 | .0 | .4 | .5 | .0 | | | |
| F | .2 | .0 | .4 | .5 | .7 | 1.0 | .8 | | | | |
| G | .4 | .8 | .8 | .3 | .0 | .0 | .5 | .0 | | | |
| H | .6 | 1.0 | 1.0 | .9 | .4 | .5 | .8 | 1.0 | .0 | .9 | |
| I | 1.0 | .5 | .0 | .0 | .1 | .8 | 1.0 | | | | |
| J | .7 | 1.0 | .6 | .6 | .0 | | | | | | |

2. Availability.

COBOL is heavily supported in the Army computing environment; the only qualification is that the 1974 version of COBOL that was used as the basis of the evaluation has not been widely implemented.

3. Standardization Status.

One of the strongest points of COBOL is its history with respect to standardization. The committees responsible for maintaining the language since its inception in 1959 have been responsive to the needs of users (both in private industry and the government) and computer manufacturers, with respect to the language requirements for business applications. Appendix A in the COBOL reference manual supplies a brief history of the standardization effort.

4. Industry Support.

As mentioned in the preceding paragraph, the development of COBOL enjoyed from the start active involvement on the part of the industrial community. As Jean Sammet, one of the individuals who participated in the early work on COBOL, relates [17, p. 332]:

> The most significant aspect of this entire activity is that it was the first attempt...to have a group of competitors work together with the prime objective of developing a language that would be usable on computers from each of the manufacturers. That it succeeded is a tribute not only to the hard work of the individuals actually serving on the committee but more importantly to the management of the various companies involved who were able to recognize the value of subordinating their own individual plans and specialties to the broader overall benefit of the customers. This was particularly significant because many manufacturers were beginning or had already done considerable work on developing their own "commercial" languages and these developments naturally had to be eliminated or subordinated to the committee results.

5. Availability of Trained Programmers.

Because of the widespread use of the language, both within and outside the Army, COBOL meets this criterion fairly well. The only qualification (as with the availability criterion) is that the differences between the 1974 and the previous standard imply additional costs in this area. However, these differences are, for the most part, minor, and they are well documented in [8].

6. Effects of Criteria on Costs.

a. Language Acquisition. Because of the availability of COBOL, the costs in this area should not be significant.

b. Programmer Training. Expenses in this area should also be fairly small; however, the complexity of COBOL will cause these costs to be higher than with other of the candidate languages.

c. System Development. Costs here will be significantly higher than with other, more technically suitable HOLs. The problem is that COBOL is not well matched to application areas outside of business data processing. As a result, programming costs are likely to be considerably greater than they would be for HOLs that are technically superior.

## Section X. PL/I AND THE MANAGEMENT DECISION-MAKING CRITERIA

This section presents the score for PL/I's technical merit and describes informally the language's compliance with the "non-technical" criteria pertinent to HOL selection.

### 1. Technical Merit.

The language vector for PL/I is displayed in Table IX. Multiplying this vector by the product of the Rating Matrix and the Application vector (see paragraph II.4) results in the scalar value 277.1 with respect to the possible range 0 through 541.4; the corresponding value in the range 0 to 100 (after being rounded to two significant figures) is 51.

TABLE IX. LANGUAGE VECTOR FOR PL/I

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A | .7 | 1.0 | .6 | .9 | .7 | .8 | .6 | | | | |
| B | .8 | .7 | .9 | 1.0 | .3 | .6 | .6 | .3 | .9 | .9 | .4 |
| C | .0 | .7 | 1.0 | .4 | .8 | .4 | .4 | .5 | .2 | | |
| D | .0 | .7 | .8 | .4 | .8 | .5 | | | | | |
| E | .5 | .0 | .0 | .0 | .0 | .5 | .5 | .0 | | | |
| F | .9 | .6 | .7 | .8 | .0 | .0 | .5 | | | | |
| G | .9 | .4 | .7 | .8 | .9 | .0 | .8 | .0 | | | |
| H | .8 | 1.0 | .9 | .8 | .6 | .5 | .9 | .5 | .3 | .7 | |
| I | .1 | .8 | .0 | .0 | .0 | .0 | 1.0 | | | | |
| J | .4 | .0 | .0 | .8 | .0 | | | | | | |

2. Availability.

a. The complexity of PL/I will likely limit implementations of the full language to large machines; however, it is probable that within a few years many large manufacturers will have near-standard PL/I compilers and support tools. A number of manufacturers of smaller computers are producing compilers for PL/I subsets. With the acceptance of the full proposed PL/I standard, the PL/I standards committee is now considering standardizing a general-purpose subset of the full language. The availability of a standard subset definition will encourage manufacturers of smaller machines to produce compilers for this subset.

b. PL/I is a fairly well-documented language, with a variety of reference manuals, guides, and primers available. Although these are not applicable to the language as defined by [2], but rather to earlier versions, the differences are sufficiently minor so as not to make this a problem.

3. Standardization Status.

On 9 August 1976, the standard proposed for the full PL/I language was accepted by the American National Standards Institute. The standard gives a number of previously obscure operations well-defined meanings. The standards committee has been strongly supported by the computer manufacturers and by the user community.

4. Industry Support.

A major problem with industry support for PL/I has been the size and complexity of the language; a substantial commitment of resources is required to develop compilers or support tools. However, there is a fair amount of support for subsets and "PL/I-like" languages.

5. Availability of Trained Programmers.

Outside the Army, PL/I is a widely used language and rates highly with respect to programmer availability. The language is less widely used in the Army, but the similarities to TACPOL would reduce the training needed for personnel already skilled in TACPOL. It should be noted, however, that the size and complexity of PL/I make programmer training a highly non-trivial effort.

6. Effects of Criteria on Costs.

a. Language Acquisition. Since the differences between the previous PL/I standard and [2] are relatively small, the costs in this area should not be large.

b. Programmer Training. As noted in paragraph 5 above, the complexity of PL/I results in training costs that are higher than those anticipated for the other candidate HOLs.

c. System Development. The complexity of PL/I and some of the language's basic design features (e.g., the preponderance of implicit conversions) combine to add cost in the system development area. This is especially true in the maintenance stage of the system's life cycle. We also note here that the absence from [2] of provisions for direct code disqualifies PL/I from consideration (unless the language is modified), since such a facility is an absolute requirement for the area of embedded computer applications.

## Section XI. SUMMARY OF CANDIDATE HOLS WITH RESPECT TO COST CONSIDERATIONS

### 1. Language Acquisition.

The estimated order of the candidate HOLs with respect to anticipated costs for language acquisition is (from low to high): TACPOL, COBOL, PL/I, JOVIAL, FORTRAN, and CS-4.

### 2. Programmer Training.

The estimated order for the candidate HOLs with respect to anticipated costs for programmer training is (from low to high): TACPOL, JOVIAL, FORTRAN, CS-4, COBOL, and PL/I.

### 3. System Development.

The estimated order for the candidate HOLs with respect to anticipated costs for system development is (from low to high): CS-4, PL/I*, JOVIAL, TACPOL, FORTRAN*, and COBOL. It should be noted that the difference between CS-4 and PL/I is expected to be significant, compared with the differences between PL/I and the remaining HOLs.

### 4. Conclusions.

Considering that over the period in which a common HOL would be used the costs associated with system development far outweigh the language acquisition and programmer training expenses, we are led to the conclusion that a technically superior language is preferable to an existing but inferior one. As a result, among the six languages considered in this report, CS-4 is recommended for selection. This conclusion will be seen to be supported by the detailed evaluations described in the remainder of this report.

---

* Assuming the inclusion of a direct code facility.

# CHAPTER 3

# TACPOL EVALUATION

## Section I. LANGUAGE SUMMARY

### 1. Lexical Properties.

TACPOL is a free-format language based upon the ASCII character set. For purposes of token formation, a fifty-character subset is employed [3, p. 32]; characters outside this set may be used in special contexts such as comments and literal character strings. The space character is significant in separating tokens, and all keywords are reserved. Identifiers may be of arbitrary length, but only the first five and last three characters are significant. The underscore may be used as a "break" character in identifiers (but is not in the fifty-character set). The comment convention in TACPOL is that used in PL/I: comments are delimited by "/*" and "*/".

### 2. Data Types.

The data types available in TACPOL can be categorized according to Figure 2.

a. Scalars. The four scalar types are short numeric, long numeric, character string, and bit string. Type checking as it exists in TACPOL is enforced only for these types; e.g., it is illegal to assign a short numeric value to a long numeric quantity. (The term "quantity" is used with a fairly specific meaning in TACPOL. As stated in [3, p. 29], "A quantity is a unit of storage employed to hold (store) a value.")

(1) Behavioral properties common to all scalar data types.

(a) Assignment and relations are defined for each scalar data type, and for no other data type. Thus, if q is a quantity having one of the four scalar types, and v is a value of some type, then the assignment

```
q = v;
```

is legitimate if and only if v has the same type

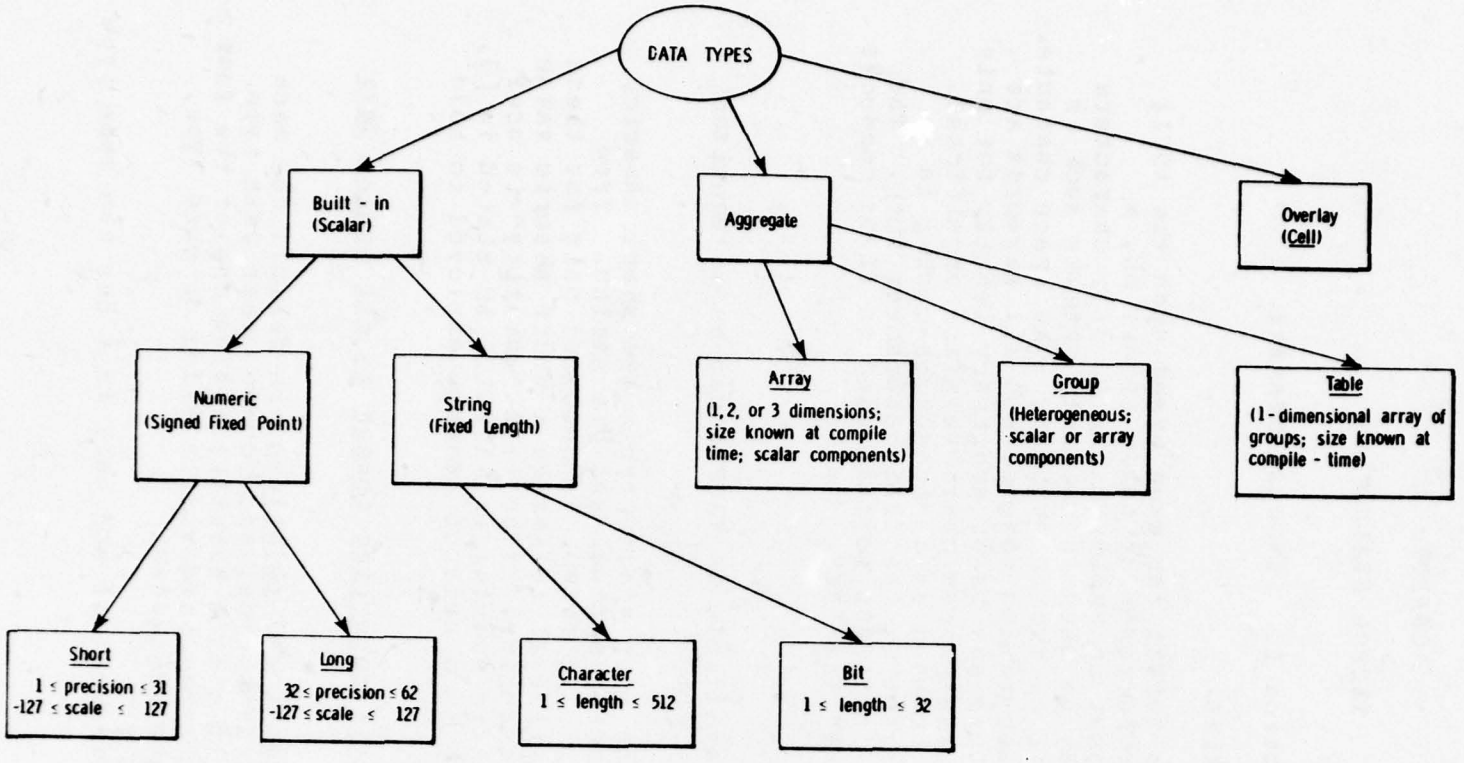



Figure 2. Data Types in TACPOL

as q. In the case of numeric operands, assignment may result in truncation of high-order and/or low-order bits, depending on precision and scale factor. For string operands, assignment when lengths differ will result in either truncation of rightmost characters or bits, or padding on the right with blanks or zeroes.

(b) The semantics of value parameters to procedures, of returning the results of function procedures, and of initialization of constants (in value declarations), are based on that of assignment.

(c) It may be noted that only scalars may be used as value parameters, function procedure results, or as constants.

(d) The relations which are defined for each scalar type are =, NE, GE, LE, GT, and LT. Though not explicitly stated in [3, Section 10.7.4], it appears that, for the numeric types, truncation may be performed during scale factor alignment. For string arguments, the shorter operand is padded on the right with blanks or zeroes before the comparison. The result of any relation is a bit string of length 1.

(2) Behavioral properties specific to the numeric types.

(a) The standard assortment of arithmetic operators are available in TACPOL. Unary + and - take a numeric operand and produce a value of the same type. Binary +, -, *, /, ** take two numeric operands (both short or both long) and produce a value of the same type. Exponentiation is restricted: only (non-negative) integer literals are allowed as exponents. Two additional operators allow the direct production of a long numeric value from short numeric operands: (*) and (**). The order of evaluation of operands in an expression is unspecified.

(b) Implicit scale factor and precision management are performed by the compiler for arithmetic

expressions. In the process, truncation of high-order and/or low-order bits may occur.

(c) A variety of utility routines (called "intrinsic procedures" in TACPOL) are defined on numeric data. These perform trigonometric and logarithmic functions, rescaling, etc.

(3) Behavioral properties specific to the string types.

(a) The CAT operator and the SUBSTR function apply to character or bit string data. The CAT operator produces the concatenation of the two operands (both of which must be either character strings or bit strings), truncating from the right end any characters (bits) which cause the result to be longer than 512 (32). The SUBSTR function (which produces a substring) may appear either on the left side or right side of an assignment. Its arguments are a string, an expression which evaluates to an index into the string, and a positive integer literal defining the length of the resulting substring.

(b) In addition to CAT and SUBSTR, there are several logical operations which apply only to bit string data (and not to character strings). These are OR, AND, and NOT; each produces a bit string result derived from a bit-by-bit computation on the argument(s) (the shorter operand is filled with Ø's on the right before the operation). There is no short-circuiting of these operations; the order of evaluation is unspecified.

(4) Explicit conversion procedures. TACPOL's "redefinition procedures" allow a value of any scalar type to be converted to a value of any other scalar type. This may involve either an actual conversion process (e.g., long numeric to short numeric) or simply a reinterpretation of the existing bits without conversion (e.g., short numeric to character string or bit string).

b. Aggregate Data Types. TACPOL contains some facilities for defining homogeneous or heterogeneous data structures; viz., the array, group, and table generators.

(1) Properties common to all aggregate data types. Component selection is the only behavioral property common to all aggregate types; assignment and relations are not defineed for these types. A representational property of aggregate types is packing: the user may specify PACKED (to minimize data space) or ALIGNED (to minimize accessing time).

(2) Arrays. TACPOL permits the specification of one-, two-, and three-dimensional arrays of scalars; the size of each dimension is given by a positive integer literal appearing in the declaration. Subscripting forms are used to select contiguous sub-arrays or individual scalar components.

(3) Groups. A group is a heterogeneous data structure consisting of scalar or array components, each having a name. This name itself is used to select the corresponding component of the group.

(4) Tables. A table is a one-dimensional array of groups; the size of the table is given by a positive integer literal. Subscripting the table name yields one of the groups. Subscripting a group component name yields the corresponding component in the selected group.

c. Untyped Data. TACPOL contains several facilities which defeat the type checking for scalar data, and the language contains no features for enforcing type checking with aggregate data.

(1) Treating data as a bitstream. The intrinsic procedures MOVE and CLEAR [3, p. 89] perform bit-transfering functions independent of the types of their arguments. MOVE(x,y) simply copies x into y as far as the shorter of the two; CLEAR(x) puts all 0's into x.

(2) Overlaying data objects. The CELL declaration has the effect of a free union; it permits instances

of scalar or aggregate types to be overlaid on the same storage area. Quantity parameter passing (i.e., "by reference") has a similar behavior. If q is a quantity passed as an argument to a procedure, where the corresponding formal parameter is a quantity parameter p, then during the execution of the procedure the effect will be that of overlaying p on the storage required for q, regardless of the data types of p and q.

3. Procedures.

a. Proper and Function Procedures. TACPOL allows the definition of "proper procedures," which do not return results; these are used in CALL statements. In addition, "function procedures" are permitted; these return scalar values as results and are used in expressions.

b. Formal Parameters. For either proper or function procedures, there are four categories of formal parameters.

(1) Value parameter. This has the effect of a "copy-in" parameter; only scalar data types are permitted. The formal parameter can be used as a local variable in the procedure (i.e., it may be assigned to).

(2) Quantity parameter. As mentioned above, this is analogous to the "by-reference" parameter in many languages, except that no type checking is performed to match the argument with the formal parameter.

(3) Procedure parameter. Parameterless proper procedures may be passed as arguments to other procedures.

(4) Point parameter. Points (i.e., statement labels) may be passed as arguments to procedures.

c. Recursion. Recursion is apparently permitted, but only for procedures which are programs.

4. Statements.

a. Null Statement. This statement allows extra semicolons to appear in programs.

b. Assign Statement. The semantics of assignment was described above (in sub-paragraph 2a(1)).

c. Call Statement. This statement is used to invoke proper procedures.

d. GoTo Statement. The TACPOL GoTo statement may be used to transfer control to any label whose name is known in the scope of the statement. Thus transfers out of procedures are permitted, but jumps into loops are not.

e. Conditional Statements.

(1) If statement. The TACPOL if-statement may have either one alternative (following THEN) or two alternatives (THEN and ELSE). The existence of one or more 1-bits in the bit-expression following IF is regarded as the "true" condition.

(2) Switch procedure. The effect of a CASE statement (or computed goto) can be obtained in TACPOL by calling an intrinsic procedure named SWITCH.

f. DO Statement. The TACPOL DO statement allows iterative execution of a sequence of statements. Two varieties of iteration are possible: stepping a loop variable by fixed amounts until a terminal value is reached, or executing the statements as long as some condition (a bit-expression) is "true". The DO statement will include one or both of these varieties. If a loop variable is used, it will receive an implicit declaration of BIN FIXED(15,0) and will thus be local to the DO statement; in addition, it may not be assigned to by the programmer.

g. Blocks. A sequence of declarations and statements may be bracketed by BEGIN and END to form a block, which may then be used as a single statement.

h. Code Statement. TACPOL allows the user to insert non-TACPOL code in the program; this is achieved via the code-statement.

5. Storage Allocation.

Storage allocation in TACPOL can be carried out statically, since the size of each data object is known at compile-time and since recursion inside programs is prohibited.

6. Process Scheduling.

TACPOL contains no facilities for process scheduling.

7. Files and I/O

a. TACPOL contains a variety of features for dealing with data files. Files may be partitioned or nonpartitioned; each partition in a partitioned file is accessed by a unique character string key. In addition, files (partitions) may be either serial (sequential access) or direct (random access).

b. There are three modes of access to a file: input, output, and update. Only direct files are permitted to be accessed for update.

c. Concurrency of program execution with data transmission during file processing is allowed in TACPOL; the default behavior, though, is to wait for the data transmission to be completed.

d. The file processing statements include OPEN, CLOSE, READ, WRITE, DELETE, SPACE, REWIND, UNWIND, and several others. File contents in TACPOL are presumed to be unstructured, binary records. No formatted I/O is available. No data type information is preserved with a file, making it possible to WRITE data of one type and to READ it back as another.

e. TACPOL provides an On-statement which allows conditional execution based on whether an indicated file condition (e.g., reading an end-of-file, accessing a record with an illegal key) occurred during a previous file processing operation.

8. Exception Handling.

Very little information is provided in [3] concerning error detection, either at compile-time or run-time. In general, erroneously obtained values (e.g., due to fixed-point overflow) are said to be "undefined", the implication being that no run-time checks are made. If it is desired to obtain such checks, condition declarations may be used to explicitly indicate this. Conditions which may be declared include zero divide, fixed overflow, assignment to a user-specified variable, and invocation of a procedure in a user-provided list. If any of the declared conditions occurs, a "snap" procedure (not defined in [3], but implied to be a trace of the block in which the condition occurred) is executed.

9. Compile-Time Facilities.

a. The TACPOL compool facility is described briefly in [6] (but not in [3]). A compool generator accepts TACPOL declarations (e.g., procedures, files, quantities) and outputs binary and symbolic data which are used as input to the TACPOL compiler. A compool serves as an outermost namescope for the program which includes it.

b. TACPOL has no facilities for macros, conditional compilation, or for the inclusion of separately written source text. A separate compilation facility is implicit in [3] but is not defined there.

## Section II. DATA AND TYPES

1. Typed Language (A1).

a. Degree of compliance: P

b. Type checking in TACPOL exists only for scalar data (short and long numeric, character and bit string), and only in the contexts of expressions and assignments. No type checking is performed when quantity (i.e., "by reference") parameters are passed. The MOVE and CLEAR intrinsic procedures treat data as untyped. The CELL facility for overlaying data objects allows type checking to be easily defeated, since no provision is made for determining which alternative is in use at run time.

c. TACPOL allows implicit declarations in the case of loop control variables, in violation of A1.

d. The changes needed to bring TACPOL into compliance with Tinman requirement A1 are fairly major, and it is doubtful whether the language could be so modified and still keep its essential character. The language facilities impacted are the data definition features, assignment, relational operators, and procedures.

e. First, if specifications of arrays, groups, or tables are to be interpreted as data types, the issue of type identity must be faced. For consistency with the scalars, the size of arrays or tables should not distinguish data type, nor should the packing (i.e., ALIGNED vs. PACKED). Groups present a problem: the simplest rule for type identity is for the field names to match and for the corresponding component types to be the same. Unfortunately, TACPOL treats field names as declared quantities and thus prohibits two groups with common field names from being specified at the same level in a namescope.

f. The assignment statement and comparison operators (for equality and inequality) should be extended to deal with data of any data type, not just scalars. The type checking would ensure, e.g., that a table was not assigned to a group.

g. Quantity Parameter passing must be made more restrictive in the interest of program reliability. It is

safe to pass an argument to a quantity parameter only if their types are the same and they are represented in exactly the same way. Thus scaling and precision (for numeric data), size (for strings, arrays, tables), and packing are relevant attributes which should be checked.

h. Implications of providing a discriminated CELL feature are discussed in A7 below.

2. Data Types (A2).

a. Degree of compliance: P

b. First, there is no floating point data type. Second, there is no explicit Boolean type. Boolean is a special case of a bit string (length 1), a situation which is not always desirable: e.g., AND and OR should not be short circuited for bit string, but they should be short circuited for Boolean (see also B6). Third, the array, group, and table facilities are restricted as to the number of dimensions and the attributes of their components (e.g., a group cannot be a component of another group).

c. The changes implied by A2 are non-trivial. If the language is to incorporate floating point, then the problem of differentiating between fixed and floating point literals arises. Also, the issue of mixed mode arithmetic must be faced; if expressions may contain both fixed and floating point operands, then conversion rules must be stated. Introducing a separate type for Boolean may be tricky, since the relationships between it and BIT(1) can be subtle. For example, if no separate literal form is provided, then the interpretation of, say, '1'B is ambiguous. Generalizing the array, group, and table features will probably necessitate a different set of rules for accessing components, and a modified syntax for declaring data objects.

3. Precision (A3).

a. Degree of compliance: F

b. Lacking a data-type for floating point values, TACPOL fails to satisfy this requirement. The impact of adding floating point was described in conjunction with A2 above.

Under the assumption that floating point is added, the additional modifications necessary to bring TACPOL into compliance with A3 are not major, but they would cause an inconsistency between the declarations for fixed and floating point quantities. (There is no requirement in the Tinman that global specification of precision for fixed point arithmetic be supported.)

4. Fixed Point Numbers (A4).

a. Degree of compliance: P

b. TACPOL fixed point numbers are binary (and thus approximations to decimal values) instead of exact as required by A4.

c. The changes needed to bring TACPOL into compliance with A4 are relatively minor; the syntax should be changed from BIN FIXED(precision, scaling), and the interpretation of literals and the results of arithmetic operations should be modified appropriately. It should be pointed out, though, that such changes have several undesirable effects. First, run-time efficiency is impaired, since scale factor alignment will (in the absence of decimal hardware) be implemented through multiplication by powers of 10, instead of simply binary shifting. Second, the rules governing literals will be made more complicated, in view of the fact that short and long numeric are distinct types in TACPOL. For example, some literals with (decimal) precision 10 will be short numeric, while others will be long.

5. Character Data (A5).

a. Degree of compliance: P

b. A5 requires that character sets be treated as any other enumeration type, and TACPOL does not contain features for defining enumeration types. However, the relational operations are defined for character data, so that some of the effects of treating character sets as enumeration types are in fact achieved.

c. The modifications needed so that enumeration types can be defined are described in connection with E6 below.

It should be pointed out, though, that attempting to obtain the character data type by such a facility is not likely to be cleanly achievable. One problem is that a single character currently is obtainable as a character string of length 1, but there is no corresponding concept of a string of objects from an enumeration type. Associated with this problem is the issue of how a character string literal can be defined, since in general there is no literal form for arrays of enumeration types.

6. Arrays (A6).

a. Degree of compliance: P

b. In TACPOL, the lower subscript bound is always 1, and the upper subscript bound must be a positive integer literal (thus known at compile time).

c. Allowing run-time (scope entry) determination of array bounds is a major change to TACPOL, necessitating a redesign of much of the data-type facility (e.g., for consistency the sizes of bit strings and character strings should also be run-time determinable). Such a change, however, cannot be carried out without defeating many of the essential objectives of TACPOL (e.g., simplicity, run-time efficiency, compile-time determination of all addresses and storage requirements).

7. Records (A7).

a. Degree of compliance: P

b. Although the CELL feature allows records of alternative structure [3, pp. 42-44], this is accomplished via a simple storage overlay technique which defeats type checking. In addition, the CELL facility restricts the kinds of data objects which may be overlaid; for example, one may not overlay a table on two contiguous groups.

c. To bring the language into compliance with A7, it would be necessary to include a "tag" field with each CELL object. A run-time check would then be generated for each reference to a CELL component, to guarantee that the current value of the tag corresponds to the referenced component.

This is a non-trivial change, and, to prevent circumventions of type-checking, impacts the procedure facility. (If quantity passing of CELL components is allowed, it is possible to obtain distinct names, with distinct data types, for the same storage.) In addition, the (implicit) generation of run-time checks is antithetical to TACPOL's goal of object code efficiency.

## Section III. OPERATIONS

1. Assignment and Reference (B1).

a. Degree of compliance: P

b. TACPOL partially satisfies B1, with the following exceptions. First, assignment is defined only for scalar types -- one cannot, for example, assign an array to another array. Second, since there are no facilities for encapsulating type definitions, there are no user-definable assignment or accessing operations. Third, since the CELL feature realizes storage overlaying, reference to a quantity will not necessarily retrieve the last value assigned to that quantity.

c. Major modifications are required to bring TACPOL into compliance with B1. Extending assignment to non-scalar data was described above in connection with A1. The encapsulation of type definition is discussed below (E5). Implications of a "safe" CELL facility were presented above in connection with A7.

2. Equivalence (B2).

a. Degree of compliance: P

b. The identity operation in TACPOL is applicable only to scalar data and is legal only if both operands have the same type; thus, TACPOL is not as general as required in B2.

c. The scope of modifications is roughly the same as that required to extend assignment to non-scalar data. This would not be a major change to the language.

3. Relationals (B3).

a. Degree of compliance: P

b. All six relational operators are in fact automatically provided for numeric data (i.e., short and long numeric), but there are no enumeration types in TACPOL. Also, shifting due to scale factor alignment may result in the loss of significant bits with no indication of error.

c. The only modifications needed are those which bring enumerated types into the language; these are discussed in E6 below.

4. Arithmetic Operations (B4).

a. Degree of compliance: P

b. Since there is no floating-point data type, the Tinman requirement that division yield a real result is not satisfied. Exponentiation is restricted in that the exponent must be a non-negative integer literal.

c. The impact of introducing a floating-point data type was described above (A2). The generalization of exponentiation is a relatively minor change but necessitates a set of special cases in the language specification.

5. Truncation and Rounding (B5).

a. Degree of compliance: F

b. TACPOL fails to satisfy this requirement. Depending on scale factor and precision, the most significant bits may be truncated during assignment, arithmetic operations, and relational operations.

c. Bringing the language into compliance with B5 represents a major change with respect to TACPOL's design goals, since it would imply run-time checking which can be prohibitively expensive in the absence of suitable hardware. This problem could perhaps be circumvented by providing facilities which allows the user to suppress the generation of the checking code, but this would add complexity to the language, again defeating one of TACPOL's goals.

d. Although the modifications needed to make TACPOL comply with B5 are significant with respect to the flavor of the language, the effects should be fairly localized with respect to the compiler. (Presumably, only the code generator phase would be affected.) Thus, the impact of the modification on an implementation is not major.

6. Boolean Operations (B6).

a. Degree of compliance: P

b. The operators AND, OR, and NOT are actually provided for arbitrary bit strings, and not simply for BIT(1). There is no NOR operator available. There is no "short-circuiting" of the AND and OR operators.

c. Introducing a NOR operator is a minor change. Providing a "short-circuiting" for AND and OR on arbitrary bit strings is undesirable, but to effect this behavior for BIT(1) and have other rules for BIT(n) where n > 1 is apt to be confusing. Alternatively, providing a separate Boolean data type, distinct from BIT(1), also raises some difficulties (see A2 above). Such problems make the introduction of short-circuiting a non-trivial design change.

7. Scalar Operations (B7).

a. Degree of compliance: P

b. TACPOL does not permit scalar operations or assignment to be performed on arrays, groups, or tables.

c. A major design change is required to bring TACPOL into agreement with this Tinman feature. The issue of extending assignment to arrays, groups, or tables was described earlier (A1); note that B7 also implies that scalar-to-array assignment be permitted. The extension of scalar operations (e.g., +, -, *, /) to composite data structures implies a data definition facility not available in TACPOL; see E5 below.

8. Type Conversion (B8).

a. Degree of compliance: P

b. TACPOL satisfies this requirement in several ways: first, the "no implicit type conversions" requirement is enforced for the operations on scalar data; second, the language provides a complete set of intrinsic procedures (called "redefinition procedures") for explicitly converting data from one scalar type to another; and third, no conversion operation is required when the type of an actual

parameter is a constituent of a union type (i.e., CELL) which is the formal parameter.

c. However, there are also major ways in which TACPOL fails to satisfy this Tinman requirement. Most importantly, an implicit type conversion of a notorious variety -- interpreting storage which contains a value of one data type as though its contents are of some other data type -- which makes program maintenance exceptionally difficult, is built into the language in its quantity parameter passage and CELL facilities. A minor instance of implicit conversion occurs in array subscripting; fractional bits are truncated to ensure integer subscripts. It should also be pointed out that the explicit conversion between numeric and character string data is not likely to be the one desired by the user. For example, the string-to-short-numeric conversion SHORT('9') produces the BIN FIXED(8, 0) value 00111001 (Base 2) (the ASCII encoding of the character '9') instead of 00001001 (Base 2) (the value 9).

d. The major design changes required to bring TACPOL into agreement with B8 have been discussed earlier, in connection with A1.

9. Change in Numeric Representation (B9)

a. Degree of compliance: P

b. The first requirement of B9 ("Explicit conversion operations will not be required between numerical ranges") is partially satisfied: such conversions are not required when source and target are both short or both long, but they are required when one is long and the other is short.

c. TACPOL fails completely with respect to the second part of B9 ("There will be a runtime exception condition when any integer or fixed point value is truncated"). No such exception condition exists: TACPOL specifies that the result of an operation is "undefined" when such truncation (of high order bits) occurs [3, p. 59].

d. To satisfy cleanly the first requirement of B9, TACPOL should not distinguish at the language level between short and long numeric data types. This is not a major design change.

e. With respect to the second part of B9, the main modification to the language would be to make explicit that "undefined" results in fact are errors which are checked at run-time. The impact of such a change, on design philosophy and compiler implementation was discussed above in connection with B5.

10. I/O Operations (B10).

a. Degree of compliance: P

b. TACPOL contains facilities for dealing with serial and direct-access files comprising unstructured binary records. No features are available (without dropping into MOL) for handling channels or devices. No dynamic assignment of I/O devices is possible. A limited capability exists for user handling of exception conditions (e.g., reading end of file, attempting to access a record with an illegal key). No data type information is associated with files; it is possible to write information of one type and read it back as another type.

c. No formatted I/O is available in the language; explicit editing procedures are necessary.

d. The language places restrictions on the allowable sources and destinations for READ and WRITE which make it illegal to READ a record into, or WRITE a record from, a quantity specified by subscripting an array; it is also illegal to WRITE a character string literal.

e. The implication of B10 is that files, channels, and devices should be definable as data types via the language's type encapsulaion facility. TACPOL lacks such a facility; it is a major design change to introduce this into the language (see E1 below). The other changes needed to bring TACPOL into compliance with B10 are less far-reaching.

11. Power Set Operations (B11).

a. Degree of compliance: P

b. TACPOL does not contain or permit the definition of enumeration types; thus, it also does not provide for power sets of these types. However, using the bit string data type the programmer can simulate part of the behavior of power sets.

c. The main change needed to bring TACPOL into compliance with B1 is to allow enumeration types in the language (see E6 below); the provision of power set types is then not a major problem. However, the resultant redundancy between power sets and bit strings is not especially desirable.

## Section IV. EXPRESSIONS AND PARAMETERS

1. Side Effects (C1).

a. Degree of compliance: F

b. TACPOL explicitly leaves undefined the order in which arguments of an expression (or arguments to a procedure) will be evaluated [3, Sections 10.7.1 and 10.4.8h].

c. It is a relatively minor and localized change to the language to define a left-to-right order for the evaluation of arguments producing side-effects. On the other hand, such a change has a large impact on the compiler if the intention is to preserve the object code efficiency which is facilitated by undefined order of evaluation.

2. Operand Structure (C2).

a. Degree of compliance: PT.

b. As evidenced in its syntax for expressions, TACPOL conforms almost completely with Tinman requirement C2; the exceptions are the following:

(1) In TACPOL, unary operators + and - have higher precedence than exponentiation. As a result, -5**2 would be interpreted as (-5)**2. A more natural interpretation is -(5**2).

(2) The concatenation operator (CAT) has low priority with respect to the other string operators. Thus '11'B CAT '011'B OR '10101'B is equivalent to '11'B CAT ('011'B OR '10101'B). Since such an interpretation is not "obvious to the reader" nor "widely recognized," explicit parenthesization should be required.

c. The modifications needed are minor. However, changing the precedence of operators in a language is generally inadvisable from a program maintenance viewpoint.

3. Expressions Permitted (C3).

a. Degree of compliance: T

b. In any context where both constants and references to variables of a given type are allowed, TACPOL also permits expressions of that type to appear. For example, there is no restriction on the form of subscripts for tables or arrays (any short expression is allowed).

c. Since C3 is independent of the other Tinman characteristics, satisfying C3 does not cause a conflict with other requirements.

4. Constant Expressions (C4).

a. Degree of compliance: F

b. TACPOL contains many features which require constants: e.g., dimension sizes for arrays and tables, precision and scaling for numeric data, string length specification, repetition factors preceding string literals, initialization in value declarations, exponent operand in exponentation, element count argument to SUBSTR. In none of these cases are constant expressions permitted.

c. The provision of a facility for interpreting constant expressions before run-time has direct impact on compiler costs, but does not affect the language design in a major way. It should be noted that compile-time (as opposed to load-time) evaluation is implied, since the constant expression value is needed to determine data storage requirements and/or object code.

5. Consistent Parameter Rules (C5).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement, with the following special cases:

(1) Only scalar data types are permitted as VALUE parameters.

(2) Only parameterless proper procedures can be passed as arguments.

(3) Only "quantity designators" are permitted to be passed as arguments to quantity parameters or to be used in READ or WRITE statements, which make it illegal to READ, WRITE, or pass an array component as a quantity argument, or to WRITE a literal value.

(4) "Only quantity and value arguments can be arguments to a procedure which is a program" [3, p. 47].

c. Allowing arbitrary data types for VALUE parameters entails extending assignment to non-scalar data; this also falls under the modifications described above with respect to A1.

d. Permitting arbitrary procedures to be passed as arguments will make the language and compiler somewhat more complex, but does not impact other features in the language.

e. Allowing subscripted quantities to be passed as quantity arguments is a minor change; similarly, generalizing the data source in WRITE statements to be expressions is a minor modification.

6. Type Agreement in Parameters (C6).

a. Degree of compliance: P

b. TACPOL satisfies this requirement only to a limited extent. Type checking is enforced for value parameters, which follow the rules for assignment, but is completely lacking for quantity parameters. An additional discrepancy with C6 is that in TACPOL the size and subscript range for array parameters must be known at compile-time instead of being run-time computable.

c. Extending type-checking to quantity parameters, and permitting run-time determination of array sizes are both major changes to the language and the implementation; their implications were described above in connection with A1 and A6.

7. Formal Parameter Kinds (C7).

a. Degree of compliance: P

b. The "constant" form of parameter, described in C7, is partially matched by TACPOL's value parameter passing. The differences are that TACPOL restricts such parameters to scalar data and allows the formal parameter to be modified (like a local variable) in the body of the procedure.

c. The "renaming" form of parameter corresponds closely to TACPOL's quantity parameter (except for the latter's absence of type checking). The inability to pass array components as quantity parameters is a special case in the language which seems to serve no useful purpose. Also, having quantity parameters (instead of value parameters) as the default appears to be the wrong choice, since hard-to-detect errors can result if the programmer is not conscious of this default.

d. TACPOL allows statement labels (termed "points") and parameterless proper procedures to be passed as arguments. A more general facility for passing procedures is required for C7.

e. The Tinman requirement that "exception handling control parameters will be specified on the call side only when needed" is not met. There is no facility in TACPOL for having optional parameters.

f. Most of the larger modifications needed to bring TACPOL's parameter passing facility into compliance with C7 have been described above. E.g., extending VALUE parameters to non-scalar data in a fashion consistent with the language implies a generalized assignment operation (A1, B1); type checking on quantity parameters was discussed in connection with A1. The provision for a more general procedure-parameter passing facility adds complication to the language but represents a fairly localized change; it is assumed that (for type checking purposes) the types of the parameters and (of a function) the result type will be required to be specified for procedures which are formal parameters. The Tinman requirement for "a formal parameter class specifying the control action when exception conditions occur" is unclear.

8. Formal Parameter Specifications (C8).

a. Degree of compliance: P

b. Specification of all attributes of formal parameters is always required (i.e., is never optional). TACPOL contains no generic procedure facility of the variety envisioned in C8.

c. The capability described in C8 appears basically to be a text substitution macro facility. The addition of such a capability to TACPOL has minor impact on the rest of the language and has a localized but non-trivial effect on implementation.

9. Variable Numbers of Parameters (C9).

a. Degree of compliance: P

b. In order to support such a facility, TACPOL requires a more general array generator (i.e., components should be able to have arbitrary type and not just scalar). Also, procedures must be able to take array parameters whose size may vary. Type checking during parameter passing is implicit in C9; implications of type checking for quantity parameters were described above (A1). In summary, the language and implementation modifications needed to bring TACPOL into compliance with C9 are non-trivial.

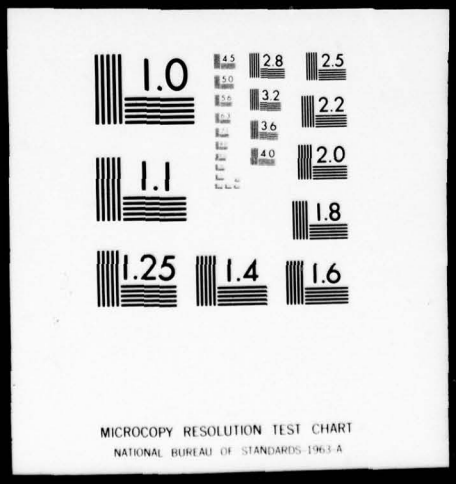
AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

14 OF 28
ADA037634

14 OF 28

ADA037634



## Section V: VARIABLES, LITERALS, AND CONSTANTS

1. Constant Value Identifiers (D1).

   a. Degree of compliance: P

   b. TACPOL partially satisfies this requirement. The INIT attribute (*) in value declarations [3, Section 10.4.7] is used for declaring constants, but applies only to scalars. Also, since the values specified are limited to literals, one cannot declare numeric constants whose values are negative (literals do not include a sign).

   c. The language changes implied by this requirement are the introduction of a facility for compile-time evaluation of expressions, and the generalization of assignment to operate on non-scalar data. The former was discussed above in connection with C4, and the latter was described under A1.

2. Numeric Literals (D2).

   a. Degree of compliance: PT

   b. As required by D2, TACPOL provides literal forms for all the built-in data types. However, there are several problems with the way the literals are realized. One is a notational inconvenience: bitstring literals must be written in binary, whereas octal and/or hexadecimal would be more readable. Another problem (unavoidable in the presence of binary fixed-point) is the mixture of binary and decimal concepts in numeric literals [3, Section 10.5.1], which is likely to cause programmmer confusion.

   c. It is a trivial change to add octal and/or hexadecimal literals to TACPOL; a notation illustrated by '716'O or '5B4'X is sufficient.

3. Initial Values of Variables (D3).

   a. Degree of compliance: F

---

(*)"INIT" is an unfortunate keyword to use in this context, since it suggests initial (instead of constant) values.

b. TACPOL has no provision for specifying initial values with variables as part of their declarations; as stated in [3, p. 30], "At the time that a quantity acquires an identity, it also acquires an undefined value."

c. The language changes needed to bring TACPOL into compliance with D3 are as follows. First, the INIT attribute should be changed in semantics so that it is used for initializations instead of constant definitions. Second, forms should be introduced for the construction of data aggregates (arrays, tables, groups, cells); these forms will be usable in initial value specification. Third, assignment with non-scalar data should be permitted, so that the rules are consistent whether scalar or non-scalar objects are initialized. These represent moderate changes to the language.

d. The impact on implementation is major if run-time testing for initialization is to be carried out. One possible technique is to have distinct modes of program compilation ("debugging" vs. "production"). A program compiled in "debugging" mode would have each of its data objects represented with an initialization tag (perhaps a single bit) which could be interrogated on each reference. The difficulty with such a scheme, however, is that it interacts unfavorably with separate compilation; handling mixtures of "debugging" and "production" modules can be non-trivial.

4. Numeric Range and Step Size (D4).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement. Since precision and scaling are provided (either explicitly or by default) for each numeric type specification, the range and step size are derivable from these values. However, ranges are not arbitrary (*) ; with precision p and scale factor s,

---

(*) It should be pointed out that there is a contradiction between TACPOL references [3] and [6] concerning the sign of numeric data. It is stated in [3, p. 28] that "all numeric values are signed." However, [6, p. 2-1] asserts: "All numeric values with eight or more bits of precision are signed. All numeric values with seven or fewer bits of precision are unsigned."

the range is -(2**p - 1)/(2**s) through +(2**p - 1)/(2**s). Also, the Tinman provision that "range and step size specifications will not be interpreted as defining new types" is not satisfied by TACPOL; short and long numeric are distinct types which differ solely in the size of the precision. Moreover, the utilization of range specifications in TACPOL for degugging purposes is fairly limited, since implicit truncation is the rule when out-of-range values are assigned.

c. The changes needed to bring TACPOL into compliance with D4 vary in scope. The addition of arbitrary ranges, with run-time bounds checking, does not have major impact on implementation but does change the flavor of the language; similar remarks hold regarding the elimination of implicit truncation (see also B5 above). Removing the distinction between short and long numeric types, on the other hand, has minor impact on the language but causes non-trivial implementation difficulties. The problem which arises is that data objects of the same type would have different representations, thereby complicating such facilities as quantity parameter passing.

5. Variable Types (D5).

a. Degree of compliance: P

b. TACPOL contains a variety of restrictions which prevent it from satisfying D5. The components of arrays may only be scalars, and arrays are limited to three dimensions. Also, group and table components may only be scalars or arrays, limiting the forms of data structures directly definable in the language. In addition, the cell feature has restrictions which prevent it from supplying a general overlay facility (see A7 above). The absence of enumeration types also detracts from TACPOL's compliance with D5.

c. The main changes needed to bring TACPOL into compliance with this requirement are the introduction of enumeration types (see E6) and the replacement of the aggregate data structuring mechanisms with a more general facility. Instead of the current array, group, and table features, the language should provide homogeneous and heterogeneous structuring facilities which can take components of unrestricted type. Thus, array and group would be generalized, and there would be no need for a table

facility (tables are simply arrays of groups). These changes have a major impact on implementation but do not cause adverse interactions with other language features.

6. Pointer Variables (D6).

a. Degree of compliance: F

b. TACPOL contains no facility for defining pointers. The changes needed to bring TACPOL into compliance with D6 are major, with respect to both language and implementation. In order for a pointer facility to be "safe," user-defined data types are needed (perhaps not of the magnitude implied by E1, but at least an ability to associate type names with specifications of data types). Such a facility requires a considerable amount of design work. Also in connection with safety, a system-maintained garbage collector is needed; this has adverse effects on run-time efficiency and would conflict seriously with one of TACPOL's major design goals.

## Section VI. DEFINITION FACILITIES

1. User Definitions Possible (E1).

a. Degree of compliance: F

b. TACPOL lacks the facilities needed to allow user definition of new data types and operations. The only definitional features in TACPOL are (1) a facility to define certain kinds of structured data objects (arrays, groups, tables), and (2) an ability to define callable procedures. These are not sufficient to meet the requirements of E1. Implicit in the notion of data type are restrictions on the uses of its member objects, but in TACPOL the lack of type checking for quantity parameters results in a complete freedom to use a data object in ways which ignore the object's declared structure.

c. It is unrealistic to modify TACPOL to bring it into compliance with E1, since the requirements listed can be satisfied only if the language designers make a conscious effort to include them as goals from the start. A data definition facility is a central component of any language, and has interactions with almost every other feature provided. Operator extension is somewhat more localized with respect to language design, but its incorporation would be antithetical to the simple nature of TACPOL and would complicate implementations.

2. Consistent Use of Types (E2).

a. Degree of compliance: F

b. Lacking user-defined types, TACPOL fails to meet this requirement. Moreover, TACPOL shows inconsistencies between the uses of scalar and non-scalar data types. For example, assignment and equivalence are defined only for scalars; constants, value parameters and the returned values for procedures (based on the rules for assignment) are similarly restricted to be scalars.

c. As mentioned in connection with E1, major modifications, entailing a substantial redesign effort, are needed if a data type definition facility having the scope called for in the Tinman is to be provided. On the other hand, the revisions needed to make scalar and non-scalar

types consistent are somewhat less drastic; the main change is the extension of assignment to non-scalar data (described in A1 above).

3. No Default Declarations (E3).

a. Degree of compliance: PT

b. TACPOL basically satisfies this requirement; the only exception is the loop control variable of the DO statement, which has the default attributes BIN FIXED(15, 0) [3, Section 10.8.6].

c. It is a minor change, both to language and implementation, to require the specification of attributes for the loop control variable. However, this change may adversely affect efficiency, since programmer-specified attributes may prevent the holding of the variable in a register.

4. Can Extend Existing Operators (E4).

a. Degree of compliance: F

b. TACPOL has no facilities for defining new data types and hence does not permit extension of operators to such types. Analogously to E1 above, there is no way to fulfill this requirement without drastic changes to the structure of the language.

5. Type Definitions (E5).

a. Degree of compliance: F

b. TACPOL does not permit the definition of new data types as required in E5. Again, as mentioned in connection with E1, such a facility cannot reasonably be added to a language unless the original design had made provisions for it.

6. Data Defining Mechanisms (E6).

a. Degree of compliance: P

b. TACPOL partially fulfills this requirement, allowing data definition via Cartesian product (array, group, table)

and implementing power sets via bitstrings. However, the language fails to provide definition via enumeration of literal names, and it realizes storage overlaying as opposed to discriminated union.

c. The addition of enumeration types to TACPOL does not represent a major change to the language; however, there are a varity of subtle issues which must be faced which impact language and implementation. For example, whether the symbolic names comprising an enumeration list must be unique is a decision which presents a tradeoff between user convenience and language/implementation complexity. In addition, the meaning of the concept of data type in connection with enumerations is somewhat cloudy.

d. The issue of adding discriminated union was discussed in connection with A7 above.

7. No Free Union or Subset Types (E7).

a. Degree of compliance: P

b. TACPOL partially fulfills this requirement. Consistent with E7, the language does not allow subset types. However, TACPOL does contain free union in the form of the cell facility and (in effect) quantity parameter passing.

c. Removing free union entails provision of a discriminated union facility and revision of the quantity parameter passing rules. The impact of the former was described in connection with A7; the latter was discussed in A1.

8. Type Initialization (E8).

a. Degree of compliance: F

b. TACPOL has no facilities for defining new types and hence lacks provision for user specification of type-specific initialization routines. As described in connection with E1, the basic structure of the TACPOL language is inconsistent with the data definition mechanism envisioned in the Tinman requirements, and it is unrealistic to attempt to modify TACPOL into compliance.

## Section VII. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed (F1)

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement. The language provides block structure, and an object may be allocated but inaccessible within a block because its name has been redeclared there. However, there is no facility in TACPOL like "own" variables of ALGOL 60, in which an object is allocated in an outer block and accessible only within an inner block.

c. It is a relatively minor change, both to the language and to the implementation, to introduce such an "own" variable facility.

2. Limiting Access Scope (F2).

a. Degree of compliance: F

b. Lacking encapsulated type definition, TACPOL fails to satisfy this requirement with respect to limited access in a type definition. The issue of separate compilation is not directly addressed in [3], although the load statement [3, p. 82] and the references to COMPOOL [3, p. 73, pp. 125ff] imply that some such facility is available.

c. The changes implied by introducing encapsulated types are major (see E1 above). The impact of a separate compilation facility on language and implementation is also substantial. For example, if strong type checking is to be enforced between separately compiled modules, the compiler must ensure that symbol table information is retained along with object code. The interaction between separate compilaion and "tagged" data for debugging was cited above (D3).

3. Compile Time Scope Determination (F3).

a. Degree of compliance: T

b. TACPOL satisfies this requirement completely, via its namescope rules. It might be pointed out, however, that the wording of the behavior of the GO TO statement [3, p. 69]

could be improved, since it is possible to interpret the description in such a way that dynamic determination of the destination point is implied.

4. Libraries Available (F4).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement. Serving the purpose of libraries are a variety of "intrinsic procedures" provided in the language for the performance of various utility functions (such as trigonometric and conversion routines). As mentioned in connection with F2, a compool facility is alluded to but not defined in [3]; presumably, such a feature would allow incorporation of predefined procedures, programs, or data.

c. In order for TACPOL to comply with F4, a specification of the compool facility is needed.

5. Library Contents (F5).

a. Degree of compliance: P

b. The compool facility, if specified in the language, would likely satisfy the majority of points in this requirement. However, there is no indication that routines written in other languages are to be available. Provision for such routines has minor effect on the language but presents major implementation problems because of the language-specific conventions governing parameter passage.

6. Libraries and Compools Indistinguishable (F6).

a. Degree of compliance: P

b. Because of the lack of explanation in [3] concerning compools, it is not clear whether TACPOL satisfies this requirement. However, there is nothing in the language which would prevent TACPOL from complying, given a suitably defined compool facility.

7. Standard Library Definitions (F7).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement, via its file manipulation facilities; however, the absence of edited (i.e., stream-oriented) I/O is a serious drawback. In addition, the interfaces to machine-dependent capabilities other than the file system is unspecified in [3].

c. As stated in the Tinman, "there is currently little agreement on standard operating system, I/O, or file system interfaces." Because of this, it is a major undertaking to attempt to define such interfaces in a language.

## Section VIII. CONTROL STRUCTURES

1. Kinds of Control Structures (G1).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement. The sequential control structure is the normal flow of control. For conditional execution the IF statement and SWITCH intrinsic procedure may be used. The DO statement handles iterative execution. Facilities for recursion, parallel processing, and exception and interrupt handling are limited (see G5 through G8 below).

c. Discussions of the scope of modifications needed to bring TACPOL into compliance with G1 are presented below, where the individual sub-requirements are considered.

2. The GO TO (G2).

a. Degree of compliance: P

b. TACPOL's GoTo statement [3, p. 69], which allows a transfer of control to a point (label) in any lexically enclosing block or procedure, is more general than G2 requires. The modification needed to bring TACPOL into compliance with G2 is relatively minor, both to language and implementation. In fact, the change amounts to a simplification, since only the most local scope need be searched to locate a GoTo's destination.

3. Conditional Control (G3).

a. Degree of compliance: PT

b. TACPOL's IF statement [3, Section 10.8.5] and SWITCH intrinsic procedure [3, Section 10.11.6] satisfy this requirement to a high degree with the following exceptions.

(1) The IF statement is not "fully partitioned:" i.e., the ELSE clause is optional.

(2) The dispatch expression for the IF statement is a bitstring and not simply a Boolean expression.

(3) Lacking discriminated union, TACPOL does not permit selection based on values of such types.

c. It might also be pointed out, in light of the importance of the CASE-statement effect for conditional control, that the SWITCH procedure should be changed to become a separate statement; this is a minor revision.

d. The modifications implied by b(1) and b(2) above are minor. The implications of discriminated union were considered in connection with A7.

4. Iterative Control (G4).

a. Degree of compliance: P

b. TACPOL's DO statement [3, Section 10.8.6] meets this requirement fairly well: e.g., the control variable is local to the loop, entry is permitted only through the head of the loop, and the common special case of a fixed number of iterations is handled by a specialized control structure. A discrepancy between TACPOL and G4 is that the termination condition in the DO statement is only tested at the beginning of the loop. Also, there are a few drawbacks to TACPOL's DO statement: a control variable must always be specified, even in the presence of a WHILE clause; and the WHILE clause's termination condition is an arbitrary bitstring and not simply a Boolean expression.

c. The changes needed to bring TACPOL into compliance with G4 are not major. An EXIT statement such as the one in CS-4, a fairly simple addition to the language, would serve to allow the termination condition to be checked at arbitrary points in the loop. An iteration of the form DO n TIMES ... END would permit loops with no control variable. Restricting the WHILE clause's termination condition to a Boolean expression, useful for program reliability, is a simple modification.

5. Routines (G5).

a. Degree of compliance: F

b. The only recursion allowed in TACPOL is at program level; as stated in [3, p. 47]: "If during the execution of the body of a proper procedure, that procedure can itself be invoked, the procedure must be a program."

c. Providing recursion in TACPOL, although a fairly localized change with respect to the language, has a major impact on implementation. The issue of whether recursion should be supported in a language is one of the fundamental decisions which will shape implementation strategy (with respect to run-time storage management); it is not wise to consider recursion as an add-on feature to an implemented language. In addition, recursion in a programming language is generaly accompanied by other facilities which make comparable demands for run-time support and which share with recursion a realm of programming applications. Such facilities include pointers and run-time determinable array bounds. The provision of such facilities in TACPOL would basically entail a redesign of the language.

6. Parallel Processing (G6).

a. Degree of compliance: F

b. TACPOL contains no facilities for creating and terminating parallel (or pseudo-parallel) processes. The only features which take concurrency of operation into account (*) are the file handling statements READ and WRITE [3, Sections 10.10.7 and 10.10.8]; when the return option is present, data transmission occurs concurrently with the execution of the statements following the READ or WRITE.

c. The addition of parallel processing is a major change to both language and implementation. Also, the conflict between TACPOL's basic goals and the Tinman's objectives is again illustrated. In order for TACPOL to comply with the intentions of G6, a safe facility for dealing with critical regions (i.e., to prevent the occurrence of race or deadlock conditions) is needed. However, the run-time overhead implied by such a facility is counter to TACPOL's efficiency objective.

7. Exception Handling (G7).

a. Degree of compliance: P

---

(*) The ENQUEUE and DEQUEUE statements [6, Sections 8-11 and 8-12], which are relevant to the use of shared resources, are also concerned with concurrency. These statements are not included in the primary reference for TACPOL [3], however.

b. TACPOL contains only a limited set of features to support exception handling. The TACPOL ON statement, despite using the same keyword as a fairly powerful facility in PL/I, is basically a form of conditional statement which detects if a previous file-handling command resulted in any of a fixed number of situations (e.g., end-of-file encountered, record-key-not-found). Condition declarations [3, Section 10.4.10] allow the programmer to monitor certain run-time conditions (e.g., division by zero, fixed-point overflow, usage of a programmer-named entity); however, the actions taken are system-dependent and not programmer specifiable.

c. The addition of exception handling to TACPOL is a major and difficult change, with respect to both language and implementation. The problems are, first, that there is little agreement concerning the proper exception facilities to support G7; second, that the interactions with other features (e.g., parameter types, parallel processing) are subtle; and third, even programs not using the facility must pay some costs associated with it.

8. Synchronization and Real Time (G8).

a. Degree of compliance: P

b. TACPOL contains only a limited set of features in this area. The WAIT statement [3, Section 10.10.15] has the effect that "the continued execution of the program is delayed until all operations requested pertaining to the designated file (partition) have been completed." The ENQUEUE and DEQUEUE statements (see G6 above) allow synchronization via exclusive access to resources.

c. The facilites called for in G8 are refinements of the requirements of G6 and G7. The discussions there are applicable here: the modifications needed have major impact on language and implementation.

## Section IX. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics (H1).

a. Degree of compliance: PT

b. TACPOL satisfies this requirement to a high degree. However, the similarity in syntax between CELL and GROUP declarations, and the use of the INIT keyword to denote assignment of value to constants, are in conflict with the Tinman's criteria of "clarity and readability of programs."

c. The modifications needed to bring TACPOL into compliance with H1 are minor syntactic changes.

2. No Syntax Extensions (H2).

a. Degree of compliance: T

b. TACPOL satisfies this requirement completely. However, the complete absence of operator-definition facilities in TACPOL, which helps the language comply with H2, brings it into conflict with E4.

3. Source Character Set (H3).

a. Degree of compliance: T

b. TACPOL satisfies this requirement completely, containing a fifty-character alphabet [3, Section 10.3.1] which includes the 26 letters, 10 digits, and the following special characters:

+ - * / = ( ) . , ; : ' $ space

This set is a subset of 64-character ASCII. It should be noted, however, that TACPOL allows a larger set of characters in comments and non-numeric literals.

4. Identifiers and Literals (H4).

a. Degree of compliance: PT

b. TACPOL satisfies this requirement fairly well, with the following differences:

(1) The effective break character for identifiers (the underscore) is not part of TACPOL's official

character set. There is no break character for literals.

(2) Although identifiers may have arbitrary length, only the first five and last three characters are significant with respect to distinguishing one name from another [3, Section 10.3.7].

c. The modifications needed to bring TACPOL into compliance with H4 are minor lexical changes.

5. Lexical Units and Lines (H5).

a. Degree of compliance: PU

b. This requirement consists of a main characteristic ("no continuation of lexical units across lines") and an exception (inclusion of end-of-line in literal strings). TACPOL complies with the main characteristic, but whether it satisfies the exception is implementation-dependent. Specifically, end-of-line may appear in literal strings if and only if the <OTHER MARK> character category includes the end-of-line character(s).

c. The change needed to bring TACPOL into compliance with H5 is to allow end-of-line character(s) as an alternative of the <CHAR SYMBOL> category [3, p. 35].

6. Key Words (H6).

a. Degree of compliance: P

b. TACPOL does not satisfy this requirement very well. The language contains over 100 key words [3, p. 94], all of which are reserved, including the names of all intrinsic procedures and four single characters (B, E, L, S) used in literal forms. There is no strong reason why intrinsic procedure names and the above-mentioned single characters could not be removed from the list of reserved words.

7. Comment Conventions (H7).

a. Degree of compliance: P

b. TACPOL partially meets this requirement. The comment delimiters are "/*" and "*/"; thus comments will not automatically terminate at end-of-line as required in H7.

c. The modifications needed to bring TACPOL into compliance with H7 are minor lexical changes.

8. Unmatched Parentheses (H8).

a. Degree of compliance: T

b. TACPOL completely satisfies this requirement, as evidenced in its syntactic rules.

9. Uniform Referent Notation (H9).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement, with the exceptions that some data references are not legitimate procedure calls (viz., when dimension marks [3, Section 10.6.3] are used) and that some procedure calls are not legitimate data references (when the number of arguments exceeds the maximum number of dimensions allowable in a table).

c. The second exception cited in subparagraph b above is fairly easily remedied in the language: the maximum number of table dimensions should be relaxed (and specified in the language, in accordance with I4). The dimension mark issue is not quite so easily resolved, however, since such notation is a conventional form; its elimination would be in conflict with H1. Also, one of the aspects of TACPOL which helps the language comply with H9 -- the fact that group component names are known at the "top level" instead of being referenced via qualification forms -- violates the spirit of H1, since it prohibits the reuse of field names in different groups.

10. Consistency of Meaning (H10).

a. Degree of compliance: P

b. TACPOL partially fulfills this requirement, with the following exceptions:

(1) The = symbol is used for both assignment and equality.

(2) FIXED(n) has different meanings in data declarations and file declarations.

(3) Value parameters may be assigned to [3, Section 10.4.8] but value quantities may not [3, Section 10.4.7]. (This is an inconsistency in the reference manual's terminology; the syntax uses different key words in the two cases -- VALUE vs. INIT.)

(4) RETURN has different interpretations depending on whether it appears in an I/O statement or a return statement. (The use of the keyword RETURN in a function procedure's return statement is mentioned in [6, p. 3-1]. The primary reference manual alludes to the return statement [3, Section 10.4.9a] but does not describe its syntax.)

c. The modifications needed to bring TACPOL into compliance with H10 are minor syntactic changes. For example, EQ should replace = as the relational operator (this would be consistent with the other operators [3, p. 65]).

## Section X. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic (I1).

a. Degree of compliance: F

b. We interpret "default" here to mean "undefined result"; this appears to be the Tinman's intention, since I1 claims that the only alternative to having no defaults is "implementation dependent defaults with the translator determining the meaning of programs."

c. Under this interpretation, TACPOL fails to satisfy I1. Undefined conditions abound, from order of evaluation in expressions and initial values of variables to situations best regarded as run-time errors (e.g., fixed-point overflow and high-order truncation, subscript out of bounds, the results of intrinsic functions such as MAX and MIN). In addition, the reference manual is incomplete in describing certain subtle behavior and thus causes implementation dependencies. As an example, when one string is assigned to another, the manual does not specify whether the constituents are copied in ascending or descending order. Since TACPOL allows the SUBSTR function on the left side of assignment statements, the different orders may yield different results. Consider the string S with value 'ABCDE' and the assignment statement:

```
SUBSTR(S, 2, 3) = SUBSTR(S, 3, 3);
```

If copying is done in ascending order, the new value for S is 'ACDEE'; if descending, 'AEEEE'.

d. Substantial modifications are needed to bring TACPOL into compliance with I1. With respect to the language definition, the behavior of every undefined or erroneous program entity must be specified. The effect of such a modification on implementation is a degradation in run-time efficiency (and thus a conflict with J1) in light of the checking which is implied.

2. Object Representation Specifications Optional (I2).

a. Degree of compliance: P

b. Of the three kinds of defaults specified in I2 (data representations, open vs. closed subroutine calls, and

reentrant vs. nonreentrant code generation), TACPOL contains facilities for only the first. By specifying PACKED or ALIGNED on his data structure declaration, the programmer can override the language-defined default for the packing attribute (ALIGNED for arrays, and PACKED for groups, tables, and cells).

c. It is not a major change to introduce defaults for open vs. closed subroutine calls, and reentrant vs. nonreentrant code generation. There are, however, interactions between such facilities and the recursive routine capability called for in G5. For example, recursive routines must be compiled closed (otherwise an unbounded code expansion would result) and reentrant. The implication is that the rules for no defaults called for in I2 will have to have "special cases," to prevent the programmer from specifying open or nonreentrant on recursive routines.

3. Compile Time Variables (I3).

a. Degree of compliance: F

b. TACPOL contains no facilities for parameterizing the object machine configuration. The language is highly dependent on the particular hardware for which it was defined (the AN/GYK-12); these dependencies permeate the language. For example: the defining document specifies the essential attributes of the object computer [3, p. 27]; limits on such values as precision, scale factor, and string sizes are dictated by AN/GYK-12 characteristics; the absence of floating point from TACPOL is due to lack of hardware support.

c. It is not a simple change to the language to add the facilities called for in I3. The distinction between short and long numeric should be removed, but (as was pointed out in D4) this can have ramifications on efficiency. The problem is that if the environmental inquiry facility implicit in I3 is intended to be used for guaranteeing portability, a fair amount of language complexity ensues. For example, in some programming environments portability should be enforced (i.e., the use of non-portable constructs should be prevented), whereas in other environments efficiency constraints may dictate deviations from such conventions.

d. The impact of I3 on existing implementations will be sizable, unless these implementations were quite careful in encapsulating their assumptions concerning object machine characteristics.

4. Conditional Compilation (I4).

a. Degree of compliance: F

b. TACPOL has no facilities for conditional compilation. It is not a major change to add a simple capability which captures the major intentions of I4; the effects are fairly localized, with respect to both language and implementation. The difficulty lies in deciding which features to leave out, so that the added capability is in fact simple.

5. Simple Base Language (I5).

a. Degree of compliance: PT

b. TACPOL satisfies this requirement fairly well. It is not an extensible language; thus the entire language can be regarded as the "base" or "kernel." In spite of this, TACPOL is quite simple (in fact, its simplicity is one of its strongest advantages). A minor example of a duplicated feature is INIT and = in constant declarations (this was discussed above in connection with D1 and D3).

c. No major changes are implied by I5, unless one interprets this requirement as calling for the existence of extension facilities in the language. However, it should be pointed out that the simplicity of TACPOL is at the expense of program reliability; an example is the preponderance of "undefined"s in contexts which should be program errors.

6. Translator Restrictions (I6).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement. The language definition establishes limits on the number of array dimensions (three) and the effective length of identifiers (eight); however, it does not restrict the number of nested parentheses levels in expressions or the number of identifiers in programs.

c. It is a minor change to the language to specify limits as required in I6. The major problem is that of deciding what values these limits should have.

7. Object Machine Restrictions (I7).

a. Degree of compliance: P

b. The absence of floating-point from TACPOL, due to the absence of floating-point hardware on the AN/GYK-12, is a violation of I7. The implications of adding a floating-point data type to TACPOL were discussed in connection with A2 above.

## Section XI. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

1. Efficient Object Code (J1).

a. Degree of compliance: PT

b. TACPOL satisfies this requirement to a high degree, as illustrated in the following subparagraphs.

(1) The restriction that string, array, and table bounds be known at compile-time avoids run-time overhead for "dope vectors."

(2) The requirement that the "length" argument to the SUBSTR function be known at compile-time avoids run-time overhead during string assignment.

(3) The cell feature, which realizes free union as opposed to discriminated union, avoids the latter's run-time "tag" checking.

(4) The parameter passing mechanism is fairly efficient. Since only scalars may be passed by VALUE, there is no implicit copying of large tables or groups; quantity parameters can be implemented efficiently via index registers.

(5) The data representations in TACPOL are keyed to the hardware characteristics of the AN/GYK-12; e.g., bitstrings are limited to 32 bits (the word length), fixed-point precision is limited to 31 (for short numeric data) and 62 (for long numeric data).

(6) The (only) justification for the multitude of "undefineds" in the TACPOL specification is one of efficiency; there is no automatically-provided run-time code to check for such conditions as high-order truncation or subscript out-of-bounds.

(7) In short, there is no reason that a good TACPOL compiler should not generate highly efficient target programs.

c. In satisfying J1, however, TACPOL comes into conflict

with a variety of requirements oriented around language power and program reliability (e.g., A6, A7, B5, D6, G5). There are no simple solutions to this conflict.

2. Optimizations Do Not Change Program Effect (J2).

a. Degree of compliance: FU

b. Since TACPOL does not specify order of evaluation for expression constituents, optimization may change program effect. For example, consider the expression F(1)+F(2) where F assigns the argument to a global variable X and then returns this as its value. Depending on which term of the sum is evaluated first, X will have the final value 1 or 2.

c. Although this requirement pertains more to the translator than to the language, TACPOL could facilitate compliance with J2 by specifying order of evaluation for side effects within an expression. As noted above in connection with C1, however, this has potential conflicts with efficiency.

3. Machine Language Insertions (J3).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement. The CODE statement [3, Section 10.8.8] permits the programmer to descend into MOL, but allows this at arbitrary places in the program. (The Tinman requires that MOL insertions be limited to the bodies of compile-time conditional statements, which are absent from TACPOL.)

c. The effort in encapsulating machine language insertions as specified in J3 comes mainly from the addition of a conditional compilation facility; see I4 above.

4. Object Representation Specifications (J4).

a. Degree of compliance: P

b. TACPOL satisfies this requirement only in a limited fashion, through group, table, or cell definitions with packing attributes specified. It is not possible to obtain arbitrary storage layouts via these facilities, nor can the programmer define "don't care" fields or associate identifiers with machine addresses.

c. The addition of a facility for specifying the detailed machine-dependent representation of data objects is a localized change with respect to language and implementation. However, the inclusion of run-time determinable array bounds (as required in A6) and discriminated unions (A7) complicates such a facility because the programmer will have to be aware of how dope vectors and tag fields are represented by the compiler.

d. To allow the user to specify time/space tradeoffs to the translator, TACPOL could incorporate optimization directives as found, e.g., in JOVIAL. This is not a difficult addition to the language. The effect on implementation can be substantial depending on the degree of optimization which is intended by the directives. There is also the possibility for conflict between a directive and another language feature (e.g., providing a time-optimization directive for a program which includes packed records).

5. Open and Closed Routine Calls (J5).

a. Degree of compliance: F

b. TACPOL does not fulfill this requirement; the programmer has no control over whether a routine is compiled in closed or open form.

c. It is a minor change to allow an attribute to be specified in a procedure declaration which will indicate open vs. closed compilation. The default should be closed compilation.

## Section XII. PROGRAM ENVIRONMENT

1. Operating System Not Required (K1).

a. Degree of compliance: P

b. Several facilities provided by TACPOL imply the existence of run-time support which is generally provided by an operating system. Condition declarations [3, p. 49] require that zero-divide and fixed-overflow exceptions be trapped, so that a "snap" procedure may be invoked. The file processing features [3, pp. 73 ff] require supervisory routines to ensure efficient and secure use of resources.

c. Bringing TACPOL into compliance with K1 would involve removing condition declarations and file processing from the language. However, that would be in conflict with other Tinman characteristics (B10, G7).

2. Program Assembly (K2).

a. Degree of compliance: U

b. The Compool facility would be useful for supporting this requirement; however, this feature is not documented in [3]. The fact that a program must be a procedure [3, p. 73] implies that some kind of separate compilation facility was likely envisioned for TACPOL, but this is not described in [3].

c. The changes needed to bring TACPOL into compliance with K2 are not major from a language viewpoint, since their effects are relatively localized and they need not be complex. However, as noted in the Tinman, implementation impact may be sustantial (e.g., enforcing type checking across the interface between separately compiled programs).

3. Software Development Tools (K3).

a. Degree of compliance: U

b. This requirement is not addressed in [3], except for a small set of debug facilities realized in condition declarations. The provision of tools such as the ones described in K3 has a major impact on language and implementation, since language/tool environment must be designed as an integrated system.

4. Translator Options (K4).

a. Degree of compliance: U

b. The options described in K4 are not addressed in TACPOL's defining document [3]. Their incorporation would not have large impact on the language, but could have major effect on implementation; a cross-reference facility is not necessarily a simple addition to a compiler.

5. Assertions and Other Optional Specifications (K5).

a. Degree of compliance: F

b. TACPOL contains none of the facilities required in K5. It is a major change, to language and implementation, to provide such features, especially since there is no widespread agreement on the kinds of specifications which are both practical and powerful enough.

## Section XIII. TRANSLATORS

1. No Superset Implementations (L1).

   a. Degree of compliance: U

   b. This requirement is a management consideration and is not addressed in the language definition.

2. No Subset Implementations (L2).

   a. Degree of compliance: U

   b. This requirement is a management consideration and is not addressed in the language definition.

3. Low-Cost Translation (L3).

   a. Degree of compliance: U

   b. This requirement is a management consideration and is not addressed in the language definition. However, the simplicity of TACPOL makes it reasonable to expect implementations to achieve the objectives of L3.

4. Many Object Machines (L4).

   a. Degree of compliance: U

   b. This requirement is a management consideration and is not addressed in the language definition. However, the orientation of TACPOL to the charactristics of the AN/GYK-12 makes it unlikely for efficient object code to be produceable for a variety of target machines.

5. Self-Hosting Not Required (L5).

   a. Degree of compliance: U

   b. This requirement is a management consideration and is not addressed in the language definition.

6. Translator Checking Required (L6).

   a. Degree of compliance: U

b. The language definition is silent concerning whether and when the checking required in L6 is carried out. As stated in [3, p. 31]:

> Whenever a semantic definition uses the terms "such as," "must be," "may only be," "cannot be," or "is undefined," the intent is to delineate proper syntacitc constructions which have meaning in TACPOL and those which have no meaning in TACPOL. In the latter case, meaning may be attributed to those constructions elsewhere, but is not here.

7. Diagnostic Messages (L7).

a. Degree of compliance: U

b. There is no discussion of diagnostic error or warning messages in the language definition. The passage cited in connection with L6 above implies that the detection of illegal constructs is implementation dependent.

8. Translator Internal Structure (L8).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

9. Self-Implementable Language (L9).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition. It may be noted that TACPOL has sufficient facilities to enable a compiler to be written in the language, but that the absence of recursion and dynamic allocation (e.g., for arrays) and the relative lack of type checking are drawbacks.

## Section XIV. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only (M1).

a. Degree of compliance: P.

b. As an existing language, TACPOL is composed from features which are within the state of the art. The problem is that advances in the state of the art have revealed problems (with respect to program reliability) in facilities such as those incorporated into TACPOL which were previously touted for efficiency and notational freedom. It is not realistic to expect to meet the Tinman requirement that "any design or redesign [to TACPOL] which is necessary to achieve the needed characteristics will be conducted as an engineering design effort and not as a research project."

2. Unambiguous Definition (M2).

a. Degree of compliance: P

b. The defining document for TACPOL partially satisfies the requirements of M2. Although the semantics are not specified formally (as in the Vienna definition language), the execution model presented in [3], in which a language feature is explained in terms of a "rewriting rule," provides a clean and clear description. Examples appear in the explanation of the cell definition [3, pp. 43-44], the value declaration [3, p. 44], the procedure invocation [3, pp. 45 ff], and the do-statement [3, p. 71].

c. Problems with the document are: occasional incompleteness (e.g., omission of descriptions for COMPOOL and the ENQUEUE and DEQUEUE statements) and ambiguity (see the example under P3 above); lack of examples; unconventional organization of material; lack of an index and glossary of terms; lack of a summary of the syntax rules.

3. Language Documentation Required (M3).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition. Comments on the language definition document are given in connection with M2 above.

4. Control Agent Required (M4).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

5. Support Agent Required (M5).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

6. Library Standards and Support Required (M6).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

## Section XV. CONCLUSIONS REGARDING TACPOL

1. Conflicts between Objectives of TACPOL and Tinman.

a. One immediate conclusion of the evaluation presented in this chapter is that there are sizable (and, for practical purposes, insuperable) differences between TACPOL and the Tinman characteristics. The cause of these disagreements lies in the differences between the priorities of the objectives sought by TACPOL and the Tinman. For TACPOL, the primary goals were object-code efficiency and language simplicity. Portability was never a serious concern, since the language was designed to be used on (and thus to exploit the characteristics of) a particular hardware configuration (the AN/GYK-12). Although software reliability and maintainability were objectives for TACPOL, at the time of the language's design the knowledge concerning the impact of language features on programming methodology and reliability was not nearly so developed as it is today. And in view of the limited range of applications envisioned for TACPOL (tactical software, as opposed to compiler writing, for example) language power was regarded as a goal which could be compromised in the interests of attaining efficiency and simplicity (thus the prohibition of dynamic array bounds).

b. The Tinman, on the other hand, placed an entirely different set of priorities on the various language goals (assuming that the amount of attention paid in the Tinman to characteristics which support an objective reflects the priority of that objective). Program reliability and maintainability and language power are the primary objectives which a Tinman-like language would seek to attain. If the efficiency sought in the Tinman's section J is to be realized also, a heavy price in language complexity appears inevitable.

2. Summary of Major Areas of Conflict between TACPOL and Tinman.

a. Data and Types. There are several serious conflicts between TACPOL and the Tinman in this area. This is illustrated by a variety of omissions from TACPOL: there is no type checking for quantity (i.e., "by reference") parameter passing; there is no data-type for floating point arithmetic; there is no discriminated union facility. In

addition, TACPOL's requirement that array bounds be known at compile-time is in conflict with the Tinman.

b. Operations. The absence from TACPOL of formatted I/O and exception signalling on truncation, and the presence of implicit type conversions, are in conflict with the Tinman.

c. Expressions and Parameters. The lack of compile-time evaluation of constant expressions and the limited parameter passing mechanism in TACPOL are in conflict with the Tinman.

d. Variables, Literals and Constants. Deficiencies in TACPOL in this area are the absence of provisions for declaration-associated initialization of variables, the lack of a pointer facility, and the limitations on data structuring.

e. Definition Facilities. This is the area in which TACPOL fares most poorly. There are no facilities in TACPOL for encapsulating type definitions in the sense required by the Tinman. (In fact, there are no facilities for defining data types at all in TACPOL.) There is no provision in TACPOL for operator extension. TACPOL does not allow data definition via enumeration of literal names.

f. Scopes and Libraries. TACPOL has no major conflicts with these requirements, except that the semantics of the Compool construct are not explicitly specified in the language definition [3].

g. Control Structures. Conflicts with the Tinman in this area stem from TACPOL's lack of recursive routines and the absence of parallel processing and exception handling.

h. Syntax and Comment Conventions. There are no major conflicts between TACPOL and the Tinman in this area.

i. Defaults, Conditional Compilation and Language Restrictions. Conflicting seriously with the Tinman is the preponderance in TACPOL of "undefined" conditions which should be detected as errors. In addition, TACPOL does not parameterize the object machine, nor does it provide a facility for conditional compilation.

j. Efficient Object Representations and Machine Dependencies. A conflict in this area is that, since TACPOL

leaves the order of evaluation unspecified for expression constituents, optimization may change program effect.

k. Program Environment. TACPOL lacks a variety of facilities, in connection with software development tools, translator options, and assertion-like specifications, which are required by the Tinman.

l. Translators. The main conflict with the Tinman in this area is that the orientation of TACPOL to the characteristics of the AN/GYK-12 makes it unlikely for efficient object code to be produceable for a variety of target machines.

m. Language Definition, Standards and Control. These characteristics do not pertain to the language per se; thus it is not applicable to speak of conflicts between TACPOL and the Tinman in this area.

3. Unnecessary Features in TACPOL.

One of the objectives of this report is to identify language features which are not needed to satisfy (but do not conflict with) the requirements, with a recommendation as to whether such features should be kept or possibly eliminated. In the case of TACPOL, there are no features which fall directly into this category; however, the question of whether a feature is or is not "necessary" to satisfy a requirement is somewhat subjective.

4. Recommendations concerning TACPOL.

On the basis of the evaluation conducted in this chapter, we conclude that an attempt to modify TACPOL to bring it into compliance with the Tinman would be inadvisable. The many fundamental differences between the two, caused by the basically different objectives desired, would imply a substantial redesign effort and a new implementation. It is not realistic to expect the "flavor" of TACPOL to be retained in such a new language. Thus, the advantages typically cited for choosing an existing language (availability of implementations, documentation, trained programmers) would effectively be forfeited. In summary, TACPOL in its present form is not suitable with respect to satisfying the Tinman requirements, and no language which differs from TACPOL in only minor ways will be substantially better.

# CHAPTER 4

# CS-4 EVALUATION

## Section I. LANGUAGE SUMMARY

### 1. Lexical Properties.

CS-4 is a free-format language which uses a subset of the 128 character ASCII character set. The subset consists of the 94 printing characters, the carriage-return and line-feed characters, and the space (blank). The space character is significant in separating tokens. Some keywords are reserved and some are reserved in specific contexts. Identifiers can consist of letters, digits and underscores, but must begin with a letter, end with a letter or digit, and not exceed 32 characters in length. There are two comment forms. The first is determined by "%" and a new-line sequence and can contain any characters. The second is delimited by "{" and "}" and can contain lexical tokens.

### 2. Data Types.

a. Built-in Types. The data types (called modes in CS-4) which are built into the language are displayed in Figure 3. The BOOLEAN mode is used for representing logical truth values. The STATUS mode is used for representing ordered sets of symbolically named data. The INTEGER mode is used for representing data whose behavior is to be that of the set of mathematical integer values. The REAL mode is used for representing numeric data whose values may be inexact and which fall in a range wider than that supplied by the INTEGER mode. The FRACTION mode is similar to REAL except that values must lie between -1.0 and 1.0. The ARRAY mode is used for representing aggregate data whose components have identical traits. The STRING mode is used for representing data comprising sequences of ASCII characters. The SET mode is used for representing the powerset of the enumerated values of an underlying STATUS mode. The COMPLEX mode is used for representing complex numbers. The VECTOR mode is used to represent all one-dimensional sequences of REAL values. The MATRIX mode is used to represent the set of real matrices. The

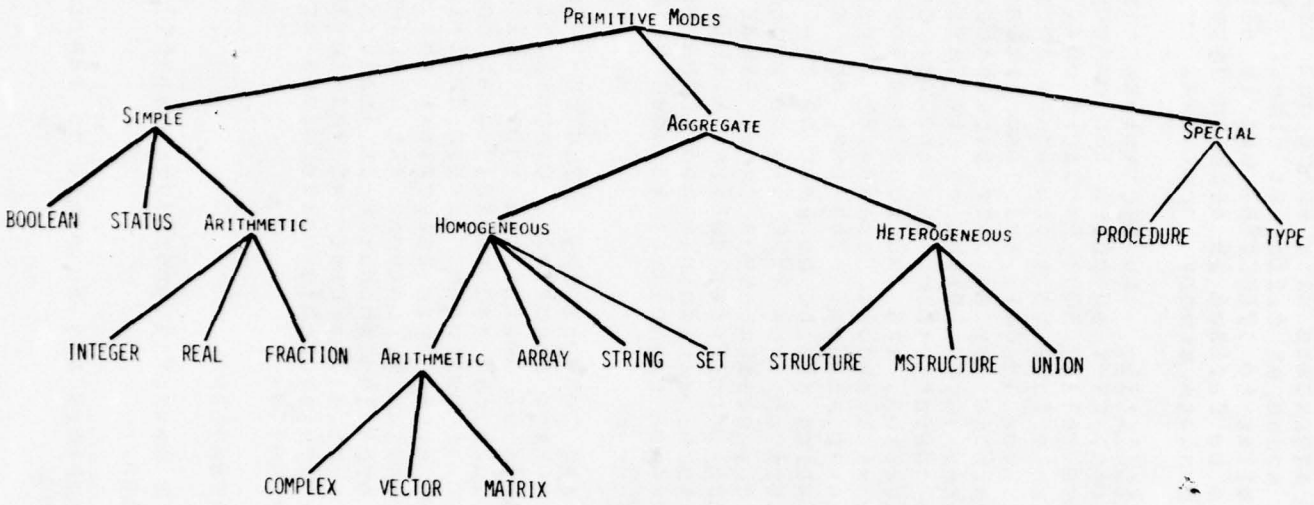



Figure 3. The Primitive Modes in CS-4

STRUCTURE mode is used for representing aggregate data whose components may be of different modes. The UNION mode is used for representing aggregate data which can take on values from different modes at different times. The MSTRUCTURE mode is similar to STRUCTURE but it allows the data representation to be defined in machine-dependent terms and/or allocated at absolute memory locations.

b. Definitional Facilities. As a complementary capability to procedures, CS-4 supports the concept of data abstraction. These are called MODE declarations. A MODE declaration is similar in form to a procedure definition. It defines a new data type (mode), and associates a name with it. MODE declarations may specify parameters required when the MODE is invoked (e.g., RANGE on the INTEGER mode). Users may specify the operations of assignment, comparison, initialization, termination, and I/O routines for defined modes. The lasnguage will supply "standard" versions of most of these if the user wishes. The user may also specify any other operations which are to be part of the interface of the mode. Parameters of type TYPE may be specified in mode declarations (e.g., element-type in an ARRAY mode). Users may specify which procedures defined within mode-definitions are to be accessible and which are to be "hidden"; this is the same capability as used with separate compilations and ACCESS.

c. Type Checking and Conversions. The CS-4 language strongly enforces data-type checking. Operations in the language accept operands of designated types only; mismatch of types is normally detected and signalled at compile-time. When a user must postpone certain decisions until run-time (for example, array size determination) the language must correspondingly postpone the completion of its checking also. CS-4 provides no built-in implicit data type conversions, and does not allow user-defined implicit conversions. Explicitly-invokable conversions are provided, and may be defined by users.

3. Procedures and Parameters.

a. CS-4 provides a powerful procedure-capability with full interface checking.

b. Procedure parameters may be bound to their arguments by COPY, REF, and NAME.

c. For full disclosure of programmer intent, for checking, and for efficiency of generated code, the parameter binding specification must designate whether the procedure wishes to obtain the value of the argument, to modify the argument, or both.

d. Keyword parameters are supported.

e. Procedure code can be expanded in line at each call (OPEN), or can be conventionally compiled (CLOSED).

f. Procedures can be passed as arguments in procedure calls.

g. Recursion is not permitted.

h. The MPROCEDURE procedure type allows definition of procedure code in target-machine assembly language. The high-level interface definition and interface checking are preserved.

i. The EPROCEDURE procedure type is a template which describes an "external" procedure written in a language other than CS-4.

4. Statements.

a. Assignment Statement. The action of an assignment statement is to invoke the special ASSIGN routine associated with the mode of the designated object.

b. If Statement. An If Statement indicates a segment of program text whose execution depends upon the run-time state of a BOOLEAN value. The ELSE phrase is optional.

c. Case Statement. A Case Statement permits the dynamic selection of one list of statements to be executed from a group of such lists. An OTHERWISE phrase is optional.

d. Repeat Statement. A Repeat Statement permits the repeated execution of a list of statements. Optional FOR and WHILE phrases are permitted to control the repetition.

e. Exit Statement. An Exit Statement is used to immediately terminate the execution of a labelled block or repeat statement.

f. Procedure Statement. A Procedure Statement is used to invoke a group of statements declared as a procedure.

g. Return Statement. A Return Statement is used to terminate execution of a procedure and return control to the point immediately following the procedure's invocation.

h. Go To Statement. A Go To Statement is used to transfer control to a labelled statement; restrictions on its use prevent such error-prone situations as jumping into a repeat-loop or out of a procedure.

i. Signal Statement. A Signal Statement is used to invoke a procedure which has been declared for exception handling.

j. Resignal Statement. A Resignal Statement is used within a signal-handling procedure to generate the same signal which caused control to pass to the current handler. The newly-generated signal will be processed by a handler which precedes the current handler on the dynamic control flow chain.

k. Abort Statement. The Abort Statement is a less restricted form of Go To Statement which may only appear in the body of signal-handling procedures and which can transfer control to a label in the immmediately containing program or procedure.

l. Begin Block. A Begin Block is used to group a sequence of declarative and executable statements together for treatment as a single compound statement.

m. Update Block. An Update Block is used to supervise the access to data which are shared by separately-executing processes.

5. Storage Allocation.

a. The storage class of an object designates the times of allocation and deallocation of the object. The scope of reference to an object never exceeds the scope of its allocation.

b. AUTOMATIC objects are allocated at block-entry and deallocated at exit.

c. STATIC objects are allocated for the lifetime of the process in which they are declared.

d. SHARED objects are allocated at the first initiation of a process in which they are declared, and deallocated at the termination of the last such process. All processes share the same object if its storage class is SHARED.

e. There are two sub-classes of sharing:

(1) SHARED(PROTECTED) objects have compiler-generated code which guarantees orderly and deadlock-free access by concurrent processes.

(2) SHARED(UNPROTECTED) objects must be protected explicitly by user code.

f. Absolute memory-location assignment is also provided.

6. Process Scheduling.

The CS-4 Operating System Interface contains facilities for creating processes and specifying priority levels or real-time constraints. In addition capabilities are provided for process-state inquiry, process interruption and resumption, process termination, shared data, and process synchronization.

7. Files and I/O.

The CS-4 Operating System Interface supports a file system. This system contains a hierarchically organized directory. Files may be stream-oriented or record-oriented and may have one of the following three organizations: consecutive, random-access, indexed sequential. In addition, CS-4 supports both edited and unedited I/O.

8. Exception Handling.

a. Comprehensive error detection is provided at compile time, program binding time, and run time.

b. A software-invokable signal mechanism, which is oriented to the dynamic procedure-calling sequence, is provided to complement the ordinary static mechanism.

c. Procedures can be designated as signal handlers. Signals can pass data to parameters of signal handlers. Type checking is performed on these parameters.

9. Compile-Time Facilities.

a. Compile-Time Evaluable Expressions. Many of the built-in operators and functions will be invoked at compile-time provided all of their arguments are known at compile-time. Such expressions can be used in contexts requiring compile-time known values.

b. Modular Composition Capability. The structuring properties of CS-4 are intended to support large-scale programming efforts involving many programmers. The unit of compilation is a PROGRAM. The interface (exportable capabilities, data objects and types, etc.) of each PROGRAM is explicit. All of the components of the interface of one PROGRAM may be accessed in another PROGRAM by means of an ACCESS statement which names the desired PROGRAM. Unlike a conventional INCLUDE directive, which fetches source text whose relationship with external procedures (etc.) cannot be checked, the CS-4 ACCESS statement causes direct examination of the compiled program itself. Thus, the checking across separate compilations is as thorough as that within a single compilation. Subsets of program interfaces may be ACCESSED as well as the entire interface. Renaming may be specified on the using side, to eliminate name conflicts. PROGRAMS may export selected capabilities which are defined internally or accessed. Thus, the program composition process can be repeatedly applied as needed to construct large-scale program structures from tractable components in a modular fashion.

c. Administrative Language Restrictability. CS-4 restricts the use of certain language capabilities. Each program must contain a declaration of any potentially dangerous features it uses. The compiler will flag any violations of these restrictions: Go To statements, Signals, Abort statements, Disabling of checking, Mstructures, Absolute allocation, Mprocedures, Eprocedures, and Non-portability.

d. Checking Directives. CS-4 normally performs a comprehensive group of checks to detect program errors. When necessary for run-time efficiency, these checks may be

selectively disabled by compile-time directive, thereby removing run-time overhead in object code size and execution time in exchange for increased vulnerability to error.

## Section II. DATA AND TYPES

1. Typed Language (A1).

a. Degree of compliance: F

b. CS-4 completely satisfies this requirement. When an object is declared in CS-4, the data type information that appears is called a mode-invocation. The mode of the object is simply the name that is invoked. The formal parameters of the invocation are called the traits of the mode. The set of values for the compile-time traits of the mode is referred to as the type that results from the mode-invocation. The template for the value of the run-time traits is called the run-time profile. Thus, both the mode and type of each object are known at compile-time. This is even true (as required by A1) for components of UNION objects, since the language always requires the assertion of which alternative of the UNION is valid.

2. Data Types (A2).

a. Degree of compliance: P

b. CS-4 satisfies this requirement almost completely. The only exception is that the language does not provide a general fixed-point facility -- the FRACTION mode (CS-4's version of fixed-point) supplies values only between -1.0 and 1.0. In addition, (although this is a fairly minor difference) there is no explicit data type for characters; a character item in CS-4 is of STRING mode with length 1.

c. The scope of modifications is discussed in paragraphs 4 (Fixed-Point Numbers) and 5 (Character Data) below.

3. Precision (A3).

a. Degree of compliance: P

b. CS-4 partially satisfies this requirement. Points of agreement are:

(1) Precision is a trait of the REAL mode; thus, real values with different precisions still have the same mode.

(2) CS-4 does permit precision specification for individual variables (In fact, the language requires such specification.)

The main disagreement between CS-4 and A3 is that the language does not allow precision specifications to be associated with a namescope.

c. The modifications required to the language are relatively minor. A global precision declaration would have to be added to the language and the precision specification currently required for individual variables would have to be made optional.

d. The impact on implementation would be relatively minor and would require changes only to the declaration processor which enters trait information in the symbol table.

4. Fixed-Point Numbers (A4).

a. Degree of compliance: P

b. The FRACTION mode in CS-4 partially fulfills this requirement. The points of difference (both of which are major) are that FRACTION values are limited to the interval from -1.0 to 1.0, and that there are no FRACTION literals. FRACTION literals can, however, be constructed from REAL literals, but need not be represented exactly.

c. The modifications required are fairly substantial. FRACTION mode was intended as a way of making REALs more efficient for the case when the exponent was zero but with similar semantics to the REAL mode. To conform to the fixed point requirement, FRACTION mode would have to be changed to define exact quantities with a range and step size specifiable by the user. This change would also impact the arithmetic operations that operate on FRACTION mode. Despite the substantial nature of the change, it can be accomplished relatively smoothly without any major impact on other aspects of the language.

d. The impact on the implementation would be substantial, affecting all phases of compilation. Assuming a well-structured compiler, however, the changes should be fairly localized within each phase, and could be made quite cleanly.

5. Character Data (A5).

a. Degree of compliance: P

b. CS-4 is defined in terms of a subset of the 128 character ASCII Standard Character Set. There is no provision in the language for the user to specify another character set. CS-4 provides the STRING mode as primitive but offers no explicit data type for characters. Although individual characters can be represented as STRINGs of length 1, this is not treated as any other enumeration type, since the only comparison operations defined for strings are equal and not equal. In order to compare strings for less than, or greater than, the built-in ASCII function would have to be used to convert the characters to their integer ASCII representation.

c. To modify CS-4 strings of length one (using the ASCII character set) to act as an enumeration type would be relatively minor. It would involve defining the less than and greater than operators for strings as well as defininining the SET mode to operate on the set of ASCII characters. To allow the user to define his own character set and also allow literals composed of strings of these characters is not within the state-of-the-art of language implementation and is also not desirable as it implies that the lexical nature of the language can change.

6. Arrays (A6).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement fairly completely. The only disagreement is that CS-4 permits both bounds of an array dimension specification to be run-time determined, while the Tinman requires the lower bound to be known at compile-time.

c. The change to the language to require the lower bound to be known at compile-time is of an extremely minor nature and is, in fact, a simplification.

d. The implementation would require a very minor change to the declaration processor.

7. Records (A7).

a. Degree of compliance: PT

b. The UNION mode in CS-4 almost satisfies this requirement completely. However, CS-4 UNIONs require a tag to be present for reliability. CS-4's HSTRUCTUREs also satisfy this requirement to a large extent, but since a tag field is not required, complete type checking cannot be ensured.

c. CS-4 cannot be modified to meet this requirement completely because the requirement itself is inconsistent.

## Section III. OPERATIONS

1. Assignment and Reference (B1).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement almost completely. As described in [4, Section 6.1.3], when defining a new mode, the programmer has the option either (1) to make no assignment procedure available to users of the mode, (2) to make available a routine specifically coded to capture the meaning of assignment for the new mode, or (3) to make available the standard compiler-supplied assignment routine. The points of difference between CS-4 and B1 are relatively minor:

(1) Assignment is not provided from a UNION component to the UNION mode and vice versa.
(2) The Tinman wants the user to "be able to declare variables for all data types." This is essentially true in CS-4; however, CS-4 has extended its concept of "mode" to include TYPE and PROCEDURE-valued parameters.

c. The modifications required for UNIONs are relatively minor. The assignment operator would have to be changed to permit an assignment from a component type to a UNION containing that type which includes changing the TAG. To permit assignment from a UNION type to a component type would require a run-time check of the TAG field for type compatibility.

d. The effect on the implementation would be relatively minor and localized to portions of the compiler dealing with UNION assignment.

2. Equivalence (B2).

a. Degree of compliance: P

b. The "=" operation in CS-4, which invokes the COMPARE routine [4, p. 164], satisfies this requirement fairly well. One exception, a minor one, is that while B2 requires the identity operation for floating-point values to check for identity only within the specified precision, the COMPARE routine for REALs in CS-4 tests for exact equality [4, p.

50]. However, CS-4 does provide a pre-defined procedure, REAL_EQ, that supplies the behavior required in B2. In addition, CS-4 does not allow "=" to compare objects of different types; such an attempt would be treated as a compile-time error.

c. One modification required is to make the REAL_EQ procedure the COMPARE procedure for REALs. For consistency, <, <=, >, >=, ~= should also be changed to invoke REAL_LT, REAL_LE, REAL_GT, REAL_GE, and REAL_NE.

d. Another modification required is to change "=" to give a FALSE value when the object types are different.

e. The impact on the implementation is fairly minor.

3. Relationals (B3).

a. Degree of compliance: PT

b. CS-4 partially satisfies this requirement. The six relational operations are always defined for types resulting from invocations of numeric (INTEGER, REAL, FRACTION) and enumeration (STATUS) modes. However, there is no way of inhibiting these operations when an ordering is not desired.

c. The language modifications required to inhibit the relationals when an ordering is not desired are non-trivial. A new, unordered STATUS mode would have to be added to the language (e.g., USTATUS). Literals for this mode would have to be differentiated from literals of the ordered STATUS mode to prevent an undesirable interaction of the two modes.

d. The effect on the implementation would also be non-trivial. Although USTATUS is similar to STATUS, extra checking is required to determine equivalence of types. In the following example

```
VARIABLE X IS USTATUS ("A", "B", "C"),
         Y IS USTATUS ("B", "C", "A"),
         Z IS USTATUS ("A", "C", "B");
```

the compiler would have to determine that X, Y, and Z all have the same type.

4. Arithmetic Operations (B4).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely [4, Section 3.2.1 through 3.2.3].

5. Truncation and Rounding (B5).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely [4, Section 3.2.4].

6. Boolean Operations (B6).

a. Degree of compliance: T

b. CS-4 provides not only the operations listed in B6 (AND, OR, NOT, NOR) but also NAND, EQV, XOR, and IMP [4, Section 3.1.4.1]. Short-circuiting is supplied for AND and OR, as well as NAND and NOR.

7. Scalar Operations (B7).

a. Degree of compliance: P

b. CS-4 does not provide the general facility required in B7. CS-4's definition of "conformable" arrays is much stricter than the definition in B7. For two arrays to "conform" in CS-4, they must have the same number of dimensions, and the same number of elements in each dimension, not merely the same number of elements as required in B7. CS-4 permits assignment and comparison between conformable arrays, but none of the other operations. Only assignment is permitted between a scalar and an array and then only in certain contexts such as array initialization. CS-4 is overly restrictive on assignments between records (STRUCTUREs) and requires the field names to be identical. CS-4 does not enforce the distinction, as required in B7, between operations defined for arrays and operations defined for modes whose object-representations are arrays. It should be noted that CS-4 supplies VECTOR and MATRIX as built-in modes that include the behavior required in B7.

c. The modifications required to the language would be fairly minor. Relaxing the restriction on STRUCTURE conformability is straightforward. Relaxing the restriction on array conformability, although simple, is undesirable as it would defeat the strong typing philosophy of the language. Extending the definition of scalar operations to conformable arrays and arrays with scalars is straightforward.

d. The effect on the implementation is minor. Relaxing the restriction on STRUCTUREs would only require the deletion of field name checking; the logical structure is already checked. Extending the operation definitions is a straightforward task.

8. Type Conversion (B8).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement almost completely with the following differences:

(1) CS-4 permits a match between an argument and a UNION formal parameter only when the types are identical. The Tinman requirement for a match when the argument mode is a constituent of the UNION mode is not met.

(2) Lacking a general fixed-point facility, CS-4 does not provide explicit conversions between fixed-point scale factors.

c. It should be noted that although CS-4 permits mixed mode arithmetic, this need not be viewed as involving an implicit conversion, but rather as instances of generics that accept arguments of different modes.

d. The modifications required to add fixed point have been discussed in connection with A4. To modify the language to permit a match when an argument is a constituent mode of a UNION mode formal parameter presents serious implementation problems when the parameter is passed by reference.

9. Changes in Numeric Ranges (B9).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely. The RANGE trait does not distinguish the mode in CS-4; when an out-of-range assignment is attempted, the X_INTEGER_RANGE or X_REAL_RANGE error is signalled [4, p. 39, p. 50].

10. I/O Operations (B10).

a. Degree of compliance: PT

b. This requirement is satisfied almost completely by CS-4's Operating System Interface [5]. The only capability not provided by CS-4 is that of allowing dynamic assignment and reassignment of I/O devices.

c. Changing the CS-4 Operating System Interface to allow dynamic assignment and reassignment of I/O devices is a minor task. The main problem, however, is that such a capability may be specifically prohibited by a particular operating system.

11. Power Set Operations (B11).

a. Degree of compliance: T

b. The SET mode in CS-4 [4, pp. 253ff], which takes a STATUS type as argument, satisfies B11 completely. However, element predicate has been implemented as the selector, which may not be a good choice since S("element"):=TRUE is not as readable as INSERT(S, "element").

## Section IV. EXPRESSIONS AND PARAMETERS

1. Side Effects (C1).

a. Degree of compliance: T

b. CS-4 completely satisfies this requirement. As stated in [4, p. 28]: "the order of evaluation of the operands within a given expression is effectively as if an operator's left operand is always evaluated before its right operand". It should be noted that this rule enables the compiler to modify the order of evaluation (for optimization purposes) when it can ensure that no side effects are involved.

2. Operand Structure (C2).

a. Degree of compliance: T

b. CS-4 totally satisfies this requirement. The only qualification is that CS-4 contains a large number of operators, and the precedence of some of these is not widely agreed upon (e.g., OUTER and CROSS for VECTORs).

3. Expressions Permitted (C3).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely.

4. Constant Expressions (C4).

a. Degree of compliance: P

b. CS-4 satisfies this requirement to a large extent [4, p. 173]. However, not all the built-in functions are compile-time invokable.

c. The required modifications are relatively minor. Built-in functions such as ABS and SGN that are not currently compile-time invokable but which could meaningfully be used in a constant expression would have to be made compile-time invokable.

d. The impact on the implementation would be relatively minor.

5. Consistent Parameter Rules (C5).

a. Degree of compliance: P

b. CS-4 does not satisfy this requirement very well. Although there is a consistent set of rules for parameters applicable to procedures, exception handlers, parallel processes, and built-in operators, there is a different set of rules for parameters to user defined modes.

c. The modifications required to meet this requirement would be substantial. Mode definitions are of such a different nature than procedures that it is not clear if the rules can be made meaningfully consistent. One way this requirement can be met is to eliminate parameters to user defined modes, although this would limit the power of the language.

6. Type Agreement in Parameters (C6).

a. Degree of compliance: PI

b. CS-4 satisfies this requirement almost completely, as explained in the rules for COPY, REF, and NAME binding [4, pp. 144-145, p. 149]. The only disagreement (also noted earlier in connection with B1 and B8) is that, in CS-4, formal parameters of a union type are not considered conformable to actual parameters of component types.

c. As discussed under B8, allowing actual parameters of a component type to conform to a formal parameter of a union type presents serious implementation problems in the case of "by reference" parameter passage.

7. Formal Parameter Kinds (C7).

a. Degree of compliance: P

b. The Tinman states that "there will be only four classes of formal parameters", including one for responses to exception conditions and one for procedure-valued parameters. CS-4 has three classes of bindings (COPY, REF, and NAME), each of which may specify that the information flow is either "in", "out", or "in-out". COPYI and REFIO nearly correspond to the first two classes of parameters required in C7 except that COPYI parameters may be assigned

to locally. CS-4 has procedure-valued parameters which must have a binding of NAMEI. CS-4 provides an exception handling control facility through a signal handler mechanism, an alternative approach to exception handling control parameters.

c. In terms of procedures, CS-4 can be easily modified to eliminate all binding classes except COPYI, REFIO and a special class for procedure-valued parameters. However, as discussed under C5, CS-4 has special rules for parameters to mode definitions (e.g., they must all be NAMEI), so making the above changes would interfere with mode definitions. Changing the exception handling mechanism to conform with this requirement would be a major change and would result in added complexity.

d. Eliminating the extra binding classes for parameters to procedures would have a relatively small impact on the implementation. However, changing the parameter classes for mode definitions and changing the exception handling mechanism would be a major change to the implementation.

8. Formal Parameter Specifications (C8).

a. Degree of compliance: P

b. CS-4 partially meets this requirement, via the MOPEN compilation class for procedures. However, although the traits of the formal parameter's mode may be omitted, the mode name itself must be specified [4, pp. 148-149].

c. It is difficult to discuss modifications to meet this requirement because of inconsistencies within this requirement as described in Appendix III. In particular the macro capability implied by the phrase "instantiated at compile-time by the characteristics of their actual parameters" does not provide the claimed "generic" capability.

9. Variable Number of Parameters (C9).

a. Degree of compliance: F

b. Since user-defined procedures always contain a fixed number of parameters, CS-4 fails to satisfy this requirement.

c. The modifications required to support this requirement would be non-trivial. The parameter passing mechanism would have to be changed to meet the requirement, possibly by augmenting it with an implicit conversion from the variable size list of parameters on the calling side to a variable size array that the procedure itself would receive.

## Section V. VARIABLES, LITERALS AND CONSTANTS

1. Constant Value Identifiers (D1).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely with its CONSTANT declarations [4, pp. 15-16]. It should be noted that CONSTANTs are run-time, not compile-time, evaluated.

2. Numeric Literals (D2).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement almost completely. The syntax and semantics for literals are explained in [4, Sections 1.3.2 (general), 3.2.1.1 (INTEGER literals), 3.2.2.1 (REAL literals), 3.3.1 (STATUS), 3.6.1 (STRING)]. The only disagreement between CS-4 and D2 is that CS-4 lacks a literal form for one of the atomic modes (viz., FRACTION).

c. The modifications required to add FRACTION literals are fairly straightforward, dependent, of course, on the extension of FRACTIONs to general fixed-point quantities (see A4).

3. Initial Values of Variables (D3).

a. Degree of compliance: T

b. CS-4 completely satisfies this requirement. In CS-4, it is necessary for the programmer to supply an initialization specification with each variable or constant declaration. However, the special NOVALUE form [4, p. 18] may be used to inhibit initialization, and (at user option) detection of erroneous usage of variables declared with NOVALUE initialization will be performed. CS-4 also meets the Tinman requirement that there be no default initial values.

4. Numeric Range and Step Size (D4).

a. Degree of compliance: P

b. Not having true fixed-point variables, CS-4 cannot completely satisfy this requirement. However, the INTEGER

and REAL modes have a RANGE trait used to denote numeric intervals, and the REAL and FRACTION modes have a PRECISION trait, which determines step size. Being traits of these modes, the RANGE and PRECISION traits do not result in new modes.

c. Modifications to add fixed-point to CS-4 have been discussed in A4.

5. Variable Types (D5).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement almost completely. The only difference is that CS-4 does not allow the definition of subsequences of STATUS (enumeration) types.

c. The modifications to permit subsequences of STATUS would be substantial and counter to the philosophy of STATUS. CS-4 permits STATUS literals to occur in any number of different STATUS types (e.g., "ORANGE", "APPLE", "LEMON" may belong to both a fruit type STATUS("ORANGE", "APPLE", "LEMON") and a flavor type STATUS("ORANGE", "CHOCOLATE", "VANILLA", "LEMON")). Consequently a subsequence such as "ORANGE.."LEMON" would be ambiguous.

6. Pointer Variables (D6).

a. Degree of compliance: F

b. Lacking a pointer facility, CS-4 fails to satisfy this requirement (it may be noted, however, that the data abstraction facility enables the user to define modes that capture some of the essential properties of pointers).

c. Modifying the language to add pointers would be a major effort,. due to interactions between pointers and other features such as storage allocation and type equivalence.

## Section VI. DEFINITION FACILITIES

1. User Definitions Possible (E1).

a. Degree of compliance: P

b. CS-4 satisfies this requirement to a large extent, particularly in its data abstraction facility [4, Chapter 6]. The only area of disagreement is that CS-4 does not allow the user to extend the meanings of infix operators to operands of user-defined modes or to define new infix operators. It should also be pointed out that a shorthand for mode-invocations (a form of renaming that does not create a new mode) would be useful, for notational convenience.

c. The modifications required to completely meet the requirement are substantial. An operator definition facility would have to be added to the language to allow the definition and extension of infix operators.

2. Consistent Use of Types (E2).

a. Degree of compliance: P

b. Although the CS-4 data abstraction facility helps remove some of the differences in use between built-in and programmer-defined modes, several differences remain:

(1) No component selection outside the mode definition is possible with user-defined modes. For the language to comply with E2, component selection would have to be invokable via either subscripting or dot qualification, and new values would have to be assignable to selected components.

(2) Some of the built-in modes have prefix or infix operators associated with them. Since CS-4 lacks a facility to define new operators or extend existing ones, no user-defined mode can have any associated prefix or infix operators.

(3) The "*" form is permitted in ARRAY mode-invocations but may not appear in invocations of user-defined modes.

(4) There are different object construction conventions for built-in vs. user-defined modes. For example, the mode-invocations for many of the built-in modes may be used as explicit generic conversion routines; such a facility is not present with user-defined modes.

c. Attempting to meet this requirement would entail major revisions to the language design. In fact, it is not clear that the state-of-the-art in language design is such that this requirement can be completely satisfied.

3. No Default Declarations (E3).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely. As stated in [4, p. 2]: "the language does not provide any default declarations for the mode of an object."

4. Can Extend Existing Operators (E4).

a. Degree of compliance: F

b. CS-4 contains no provision for extending existing operators to new data types.

c. The required modifications would be non-trivial. As mentioned under E1, an operator definition facility would have to be added to the language.

5. Type Definitions (E5).

a. Degree of compliance: T

b. CS-4 satisfies this requirement via its data abstraction facility. A MODE definition specifies both a representation (parameters to the MODE and constants and variables defined within the mode-body [4, p. 158]) and a set of routines specific to the MODE. Since a user-defined mode is automatically different from the mode of the underlying representation, there is no way for routines applicable to the latter to be inherited by the former. In fact, not even all the routines that are defined within a mode-body are usable outside; the CAPABILITY attribute [4, p. 162] provided in the mode definition defines which of the

routines are so usable. As required by E5, CS-4 permits the user to define constructors, selectors, predicates, and type conversions; however, each of these must be invoked via explicit procedure invocations.

6. Data Defining Mechanisms (E6).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely. Definition by enumeration of literal names is realized in the STATUS mode [4, Section 3.3]; Cartesian products are represented by ARRAY and STRUCTURE modes [4, Sections 3.4 and 3.5]; discriminated union exists in the form of the UNION mode [4, Section 3.7]; and the SET mode [4, Appendix B] captures the notion of the power set of an enumeration type.

7. No Free Union or Subset Types (E7).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement to a large extent. First, type definition by subsetting is not permitted. Second, although type definition by free union is possible (via MSTRUCTUREs [4, Section 8.1] or by disabling tag checking for UNION objects [4, Section 3.7.3.1]), programmers must possess special authority to achieve this effect [4, Section 7.2.2].

c. It is impossible to completely satisfy this requirement without conflicting with other requirements because of inconsistencies in the Tinman. Eliminating MSTRUCTURES from CS-4 would violate requirement J4, and always requiring run-time tag checking on UNION objects would violate J1, the requirement for efficient object code. In particular, it is desirable to eliminate run-time tag checking in the body of CASE statements where the tag was used to select the appropriate case.

8. Type Initialization (E8).

a. Degree of complaince: T

b. CS-4 completely satisfies this requirement, via the user's ability to define INIT (initialization) and TERM

(finalization) procedures as part of a data abstraction [4, pp. 165-167]. The INIT procedure is invoked implicitly at object allocation time; TERM is invoked implicitly on deallocation.

## Section VII. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed (F1).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely. First, since the allocation rules follow a stack discipline, it is impossible for an object to be accessible unless it is also allocated (i.e., an object is deallocated by the system, not by the programmer, and this can occur only when the object is no longer accessible). Second, the provision of STATIC and AUTOMATIC storage classes [4, p. 20] allows the user to distinguish between access and allocation scopes. For an AUTOMATIC variable, these scopes are identical. For a STATIC variable, the allocation scope is, in effect, the whole program, but the access scope is the block in which the declaration appears.

2. Limiting Access Scope (F2).

a. Degree of compliance: T

b. CS-4 completely satisifies this requirement, primarily via its ACCESS directive [4, Section 7.3.1], which enables a program to make use of information defined in separately compiled programs. The ACCESSed program makes available via a CAPABILITY list a set of entities (variables, constants, modes, or routines) whose definitions are known within the program; the ACCESSing program specifies those entities that it requires, and it may rename them to avoid name conflicts. It should also be pointed out that CS-4's data abstraction facility provides "the ability to limit the access to separately defined structures;" the mode definition's CAPABILITY list specifies the available routines.

3. Compile Time Scope Determination (F3).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely. The scope of identifiers is always determined at compile-time, and declarations must precede references ("all declarative statements in a statement list must precede all executable statements in the same list" [4, p. 14]. Although the

exception handling facilities of CS-4 involve dynamic scope search rules for determining signal handlers [4, Section 5.3.3], the identifier names involved still have statically established scopes.

4. Libraries Available (F4).

a. Degree of compliance: P

b. CS-4 partially satisfies this requirement, via the ACCESS directive. However, there are some drawbacks to this facility that prevent it from achieving all the goals of F4. The following description appears in [4, p. 177]:

> Accessed entities behave as if their actual declaration had occurred at their point of access. Access of an entity whose declaration depends upon another entity which is not made available via the capability list attribute in the header of the accessed program will constitute a compiler-detected error.

Under these rules, program modularization can become quite tricky. As an example, suppose a program is to be written that will create, retrieve, and update data in some data base. A direct approach is to encapsulate in one program, say P, the definition of the data base (STATIC VARIABLE declarations) and the definition of routines that massage it (PROCEDUREs), but to promote only the routines. In this fashion, the main program Q would access P to interface with the data base. Since the representation of the data base had not been promoted, Q would be restricted to performing only "safe" operations on the data.

c. Unfortunately, the rule quoted above would force the promotion of P's VARIABLE declarations as well as its routines, since the variables comprising the data base representation would naturally be used as free-variables inside the routines. Thus, either "unsafe" access to the representation would be granted, or a different form of program modularization would be required.

d. In addition to the modularization problem, the rule as stated in [4] gives a misleading impression in its first sentence that a macro-like facility is being applied. If this were the case, then ACCESSing a variable would cause a

copy of the variable to appear in the ACCESSing program -- this is not the intended behavior, however.

e. The modifications required both to the language and the implementation to clean up the ACCESS mechanism are of a fairly minor nature. The rule regarding accessed entities would have to be changed so that entities that are not promoted via the capability-list attribute, while not directly available to the accessing program, would still be available to entities in the accessed program that were promoted (rather than causing a compiler-detected error).

5. Library Contents (P5).

a. Degree of compliance: T

b. CS-4 completely satisfies this requirement. First, anything definable in the language may be ACCESSed in a (separately-compiled) program. Second, CS-4 allows the use of external procedures [4, Section 8.3] (i.e., procedures whose bodies are written in other languages).

6. Libraries and Compools Indistinguishable (P6).

a. Degree of compliance: T

b. CS-4's ACCESS facility can be used to build both libraries (containing data objects and routines) and compools (containing mode definitions); thus, libraries and compools are indistinguishable.

7. Standard Library Definitions (P7).

a. Degree of compliance: PT

b. CS-4 goes quite far in satisfying this requirement. The primary facility for achieving "standard machine independent interfaces to machine dependent capabilities" is CS-4's Operating System Interface. As stated in [5, p. v]: "The purpose of the operating system interface is to standardize the interface between CS-4 programs and the variety of target machine operating systems which provide common functions in non-standardized forms." Other facilities in CS-4 for dealing with machine dependencies include the MSTRUCTURE mode (for defining physical storage layout), a direct code capability, an external procedure facility, and an environmental inquiry mechanism.

c. Although the CS-4 Operating System Interface comes close to satisfying this requirement completely, there are some areas in which the definition of the facilities provided is incomplete or vague. For example, nowhere is it specified whether or not EVENT is a mode, nor is it specified whether or not an update block (i.e., a critical region) can contain operations that cause the process to be blocked.

## Section VIII. CONTROL STRUCTURES

1. Kinds of Control Structures (G1).

a. Degree of compliance: PT

b. CS-4 provides most of the control structures specified in this requirement. Sequential execution is the normal flow of control. Conditional execution is realized by the if-statement and case-statement [4, Sections 4.5 and 4.6]. Iterative execution is achieved in the repeat-statement [4, Section 4.7]; exception and asynchronous interrupt handling are achieved in the SIGNAL [4, Section 5.3] and event [5, Section 4.2] facilities. The only control structure required in G1 that is not contained in CS-4 is the ability to define or invoke recursive procedures. As stated in [4, p. 140]: "recursive calls (calls on a procedure while there still exists an active invocation of the same procedure) are not permitted."

c. The scope of modifications needed to add recursion to the language are discussed under requirement G5.

2. The Go To (G2).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement almost completely. The only qualification is that in CS-4 the GoTo statement may transfer control out of a BEGIN block in which it is enclosed [4, p. 124 and 135-136].

c. Only very minor modifications would be required to the language and the implementation in order to restrict the GoTo to the local scope level.

3. Conditional Control (G3).

a. Degree of compliance: P

b. The if-statement and case-statement of CS-4 partially satisfy this requirement. Differences that arise are (1) CS-4's conditional control operations are not "fully partitioned" (the if-statement need not contain an ELSE clause), and (2) there is no "general form of conditional which allows an arbitrary computation to determine the selected situation."

c. The modifications needed to require an ELSE clause on an if-statement are extremely minor. Although a general form of conditional such as Zahn's device adds no power to the language, it could be added without great difficulty.

4. Iterative Control (G4).

a. Degree of compliance: PT

b. The repeat-statement of CS-4 [4, Section 4.7] satisfies this requirement to a high degree. As specified in G4, the only loop control variable is local to the loop, entry is permitted only at the head, the common special case of a fixed number of iterations is handled efficiently, and the termination condition may appear anywhere in the loop through the use of an exit statement. Differences between CS-4 and G4 are as follows:

(1) The syntax for the fixed-number-of-iterations case is somewhat clumsy (to have a loop repeated, say, 10 times, the user must specify
FOR INTEGER (RANGE: 1 THRU 10) REPEAT...END or some other integer-mode-invocation denoting a value space with 10 elements).

(2) The case in which the number of iterations is fixed, but known only at run-time, cannot be handled with a for-phrase; instead, a while-phrase must be used, implying that a non-local loop-control variable will be available after loop-exit.

c. Modifying the for-phrase to permit a run-time bound and cleaning up the syntax are small changes to both the language and the implementation.

5. Routines (G5).

a. Degree of compliance: P

b. Lacking recursion, CS-4 fails to satisfy this requirement.

c. The modifications required to the language to add recursion are fairly minor. The programmer should be explicitly required to indicate which procedures are

recursive. This not only aids in readability but also aids the implementation in determining when an illegal (i.e., unintended) recursive call is made, and in enforcing the restriction that procedures cannot be defined within the bodies of recursive procedures. The changes required to the implementation are more substantial but are simplified by the fact that the dynamic storage allocation mechanism needed for recursive procedures is already required in CS-4 because of such features as dynamic array bounds at block entry.

6. Parallel Processing (G6).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement to a large extent. The only differences lie in the generality of CS-4 vs. the restrictions recommended (for efficiency reasons) by G6; e.g., CS-4 allows the definition of routines within the body of parallel routines that can have multiple simultaneous activations.

c. The main aspects of CS-4's parallel processing facility are as follows:

(1) Process management.

(a) Processes in CS-4 are created by declaring variables of mode PROCESS. A SCHEDULE_PROCESS command associates a program-level procedure with the process and indicates that the process is to be scheduled. The actual scheduling of processes is determined by user-specified priority levels or, for time-critical processes, by user-specified real-time constraints [5, p. 21, 25].

(b) The storage class for PROCESS objects is not restricted. Therefore, processes can be declared AUTOMATIC; these will be initiated dynamically. Nevertheless, because there is no recursion in the language, the maximum number of processes that can be created is known at compile-time.

(c) General capabilities are provided for

process-state inquiry, process interruption and resumption, process termination, and quaranteeing that an executing process will not be pre-empted [5, pp. 21-29].

(2) Shared data.

(a) A SHARED storage-class is provided for data objects that are to be accessed by several processes.

(b) Structured critical regions called UPDATE-blocks are provided to prevent disorder and deadlock that could result from concurrent accesses to shared data. All objects declared with the storage-class-attribute SHARED(PROTECTED) are only allowed to appear within such critical regions [4, p. 20, 127].

(c) A "corelink" capability is provided for the communication of data between processes having unrelated compilations [5, p. 33].

(3) Synchronization.

(a) Synchronization between processes is achieved by EVENT variables which can be SET by one process and WAITed on by other processes [5, p. 31].

(b) A user-specified connection can be established between EVENTs and implementation-defined hardware conditions. The EVENTs will be SET on the occurrence of these conditions and thus allow responses to hardware interrupts to be programmed in the high-level language [5, p. 32].

(c) Process delays based on a real-time clock are provided by the TIME_WAIT function [5, p. 67].

d. The modifications required to the language and the implementation to provide the restrictions required in G6 are of a minor nature.

7. Exception Handling (G7).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement fairly well through its signalling facility [4, Sections 4.11, 4.12, 4.13, 5.3.1], which provides a means of communicating the occurrence of an exceptional situation to user-specified handlers for that exception. Signal-handlers may be passed information via parameters, and they have rules consistent with those for normal procedures. Handlers may (1) handle the exception and return control to the point where the signal occurred; (2) decide the exception cannot be handled, take some other action, and transfer control to a point in the handler's immediate external environment; or (3) issue another signal.

c. Signals may be generated by explicit statements written in the program; in addition, they will automatically be generated for errors detected by the hardware (e.g., division by zero) or by compiler-supplied software checks (e.g., range and subscript bounds errors). The names of signals associated with errors are available to the user so that each program can define its own error-handling routines [4, Appendix F].

d. The only difference between CS-4 and G7 is that the CS-4 signal handling facility provides a slightly more general capability than that required by G7 and that the specification of the signal handler is not done through an exception handling formal parameter class.

e. Though not directly related to G7, there is a somewhat subtle interaction worth noting between CS-4's exception handling and data abstraction facilities. The problem is that the dynamic chain search rules prevent a single handler from handling a signal generated by disjoint procedures occurring within a MODE (or at program level). The reason for the problem is that at the time the signal is generated there would be no currently-active procedure that immediately contains the handler (the handler is defined local to a mode and not a procedure).

8. Synchronization and Real Time (G8).

a. Degree of compliance: P

b. CS-4's Operating System Interface contains facilities that meet many of the requirements of G8. EVENT variables [5, Section 4.2] permit delay until some specified situation has occurred. If the delay is to be determined by a known time interval, then time-critical processes may be used [5, pp. 24-25]. Specification of relative priorities among parallel control paths is realized in the SCHEDULE_PROCESS command for normal (i.e., non-time-critical) processes [5, p. 26]. EVENT variables may be connected to real-time clock or asynchronous hardware interrupts [5, p. 66]; however, this is a separate facility from the (synchronous) exception handling mechanism described in G7. The synchronization of parallel processes may be achieved via events, corelinks [5, Section 4.3], or protected variables and update blocks (See G6).

c. Problems in CS-4 with respect to G8 come not so much from failure to satisfy specific requirements (although it might be pointed out that CS-4 lacks a convenient facility for a process to terminate itself) as from the occasionally incomplete nature of the description. For example, despite the attention paid to EVENT variables, nowhere is it mentioned whether EVENT is in fact a mode, or how latched EVENT variables are initialized. Also, it is not clear why only pulsed events should be connectable to external conditions. It should be mentioned, too, that having procedures such as SCHEDULE_PROCESS and SET_EVENT return a STATUS value causes a fair amount of notational clumsiness; one must either declare an otherwise unnecessary variable of the given type, or else embed the procedure in a conditional statement.

d. A moderate amount of revision is required to "clean up" the Operating System Interface. This includes both making the specification complete and redesigning some features to avoid notational clumsiness.

## Section IX. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics (H1).

a. Degree of compliance: PT

b. CS-4 satisfies most of the requirements of H1. The language is free format, allows mnemonically significant identifiers (up to 32 characters in length [4, p. 6]), uses conventional notation, and does not permit abbreviation of identifiers or key words. The main deviation from H1 is that the grammar (at least the one presented in [4]) is somewhat complicated and even ambiguous. In connection with this, however, it should be noted that the grammar contained in [4] is intended to facilitate the description of the language; the complexity is partially caused by the desire to capture in the syntax a large part of the semantic restrictions defined in the language.

c. Although the semicolon serves as a separator rather than as an explicit statement delimiter, this causes no problems since CS-4 is a statement-oriented language, not an expression-oriented language.

d. It should also be noted that although the *-form of subscript and bounds specification is in fact a unique notation for a special case, it is also a conventional form.

e. The modifications required to meet this requirement are fairly minor. An unambiguous grammar, suitable for use as an implementation grammar, has been devised for CS-4.

2. No Syntax Extensions (H2).

a. Degree of compliance: T

b. CS-4 completely satisfies this requirement.

3. Source Character Set (H3).

a. Degree of compliance: P

b. CS-4 partially satisfies this requirement. A major difference is that the CS-4 character set, based on full ASCII, contains elements outside the 64-character ASCII subset, with no defined transliteration for some of the

inaccessible characters. The characters outside the subset are vertical-bar, grave-accent, open- and closed-brace, and tilde. The lack of a substitute character for the vertical bar can be a serious problem, since the string concatenation operator is composed from this symbol.

c. A very minor amount of modifiction would be needed to define translations from the current character set to the 64-character ASCII subset. Only the lexical analyzer would be affected by the change.

4. Identifiers and Literals (H4).

a. Degree of compliance: PF

b. CS-4 satisfies this requirement nearly completely [4, pp. 6-8]. The underscore is the break character for identifiers, but there is no break character for literals.

c. The modifications to add a break character for literals would be trivial. Only the lexical analyzer would be affected. The main difficulty, however, is in choosing a suitable character. There is no generally accepted break character for literals and none of the likely candidates (e.g., blank, underscore) is clearly superior to the others and each presents problems with readability and error detection.

5. Lexical Units and Lines (H5).

a. Degree of compliance: P

b. In CS-4, there is no way to continue lexical units across lines, but the language fails to satisfy the part of H5 that requires a means for including end-of-line characters in string literals.

c. The modification required to completely meet the requirement involves only a small change to the lexical analyzer.

6. Key Words (H6).

a. Degree of compliance: P

b. CS-4's treatment of reserved words [4, Appendix E] conforms partially to this requirement. Symbols in CS-4 are grouped into three categories: (1) those whose meanings are fixed (reserved); (2) those whose meanings may be assigned freely by a program but which have fixed meanings in special contexts independent of programmer-assigned interpretations; and (3) those whose CS-4 meanings are lost (overridden) when new meanings are assigned by programs (e.g., built-in functions). The keywords referred to in H6 correspond to all the category (1) and (2) key words in CS-4 and a few of the category (3) key words. All the category (1) key words satisfy H6 with one exception. TRUE and FALSE, which are literals of MODE BOOLEAN, are useable in identifier contexts.

c. The modifications required to meet H6 are fairly minor. Category (2) would have to be eliminated and all the category (2) key words put into category (1) (i.e., reserved). A few of the category (3) key words would also have to be added to category (1) (i.e., AND, OR, NOT, NAND, NOR, EQV, XOR, IMP). The Boolean literals TRUE and FALSE should be changed to .TRUE and .FALSE to avoid confusion with identifiers.

d. It should be noted that although CS-4 has a large number of key words (categories (1) and (2) have about 100), eliminating many of them would change the nature of the language and defeat some of the language's goals (e.g., readability and reliability through useful redundancy).

7. Comment Conventions (H7).

a. Degree of compliance: P

b. CS-4 partially satisfies this requirement [4, pp. 9-10]. H7 requires that the language provide only one comment convention. CS-4 provides two comment forms: one version (where the comment is delimited by percent-sign and end-of-line) fully meets the provisions of H7; the other form (delimited by braces) enables arbitrary program segments to be easily turned into a comment.

c. This requirement can be met simply by deleting the second comment form from the language.

8. Unmatched Parentheses (H8).

a. Degree of compliance: F

b. CS-4 satisfies this requirement completely.

9. Uniform Referent Notation (H9).

a. Degree of compliance: P

b. CS-4 satisfies this requirement only to a limited extent. The main example of uniform referent notation in CS-4 is that ARRAY subscripting and procedure invocations share some common forms. However, the *-form is available for ARRAY subscripting and not as a procedure argument, and key-word parameters are available for procedures but not for subscripting. The STRUCTURE mode violates uniform reference, since the dot qualification form used for data reference is not a valid function call. In addition, the absence of facilities for user-defined functions to be invokable in write-contexts (e.g., as the target of an assignment) implies that, with the exception of some built-in functions such as SUBSTR, forms appearing in such contexts will be data-references. The lack of a uniform referent notation is even more apparent for data abstractions, since no data reference form is available for selecting object components.

c. Modifications to the language to meet this requirement would be substantial and would, in effect, change the nature of the language. The dot qualification form of STRUCTURE can be fairly easily changed so that, for example, instead of writing S.A, one would write A(S). Similarly, keyword parameters can be eliminated from procedures. The other changes, however, are much more major. Eliminating the *-form of array subscripting would seriously hamper the power and convenience of the language. Allowing user-defined functions in write-contexts would require sizable design effort, especially to establish suitable restrictions. With regard to data abstractions, determining a uniform referent notation is currently a research topic and not within the state-of-the-art.

10. Consistency of Meaning (H10).

a. Degree of compliance: T

b. CS-4 complies with this requirement completely. In particular, the error-prone features cited in H10 (use of = to denote both assignment and equality, special interpretation for parenthesized arguments) are absent from CS-4.

## Section I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic (I1).

a. Degree of compliance: P

b. The CS-4 language specification suffers from a problem common to many language reference manuals, which brings it into conflict with I1. The difficulty lies in the inconsistency with which the behavior of erroneous programs is treated. In many cases, the language defines run-time conditions that are signalled when errors occur [4, Appendix F]. However, there are many more instances in which rules are specified that do not define the program behavior when the rules are violated: e.g., the restrictions governing update blocks [4, p. 128], go-to statments [4, p. 136], or even what happens with programs containing syntax errors. The result of such an absence of specifications is, as stated in I1, "implementation-dependent defaults with the translator determining the meaning of programs."

c. A moderate amount of revision is necessary to the language specification to explicitly state the program behavior when rules are violated. Such specification could be achieved by a formal definition of the language, although the latter would be a major undertaking.

2. Object Representation Specifications Optional (I2).

a. Degree of compliance: P

b. CS-4 partially satisifies this requirement. With respect to data representations, the programmer may specify storage formats only via the MSTRUCTURE mode; in this sense, he may override the default representation chosen. With respect to subroutine calls, the default is closed (i.e., out-of-line) compilation, but this may be overridden by the programmer. There is no programmer control over reentrant vs. nonreentrant code generation; this decision is implementation dependent. A complete list of compiler-supplied defaults in CS-4 appears in [4, Appendix D].

c. The modifications required for the language to give the programmer explicit reentrant code generation are minor. Moderate modifications might be required to an implementation in order to generate reentrant code, and there are potential conflicts with efficiency if the hardware does not support reentrant procedure calls.

3. Compile-time Variables (I3).

a. Degree of compliance: PT

b. CS-4 almost completely satisfies this requirement. CS-4 has an environmental inquiry facility. In addition to language defined constants, this facility also has machine-dependent constants which can be interrogated inside a program [4, Section 8.4]. In addition, the language provides for machine-dependent machine-names and storage-unit-names for specifying the machine configuration in #STRUCTURE definitions. The only way in which CS-4 fails to meet I3 is that environmental inquiry is not specified for such facilities mentioned in I3 as operating system, peripheral equipment, and special hardware options.

c. The modifications required to include additional environmental inquiry constants to the language are minor.

4. Conditional Compilation (I4).

a. Degree of compliance: P

b. CS-4 partially satisfies this requirement. There is no compile-time conditional facility, but the language does contain an environmental inquiry capability [4, Section 8.4] and allows some expression evaluation at compile-time.

c. Substantial modifications would be required both to the language and the implementation to add a conditional compilation facility. A significant amount of design would be required to select the appropriate forms of compile-time statements (e.g., CTIF, CTCASE), compile-time variable definitions and compile-time expressions. A non-trivial amount of effort would also be needed to implement such facilities.

5. Simple Base Language (I5).

a. Degree of compliance: P

b. CS-4's base language is not very simple, (comprising almost all of [4] and [5]); moreover, the reference manual does not distinguish between base language and extension-obtainable constructs. However, the latter are identifiable implicitly, consisting of many of the built-in procedures for the primitive modes (e.g., [4, Appendix C]).

c. One aspect of CS-4 that contributes to its lack of simplicity is the inclusion of COMPLEX, VECTOR and MATRIX as built-in modes with built-in operators. This was done for notational convenience since the language lacks an operator definition facility. The major complexity of the language, however, comes from the data abstraction facility [4, Section 6] which is specifically needed to meet requirements E1 and E5. A large part of this complexity comes from interactions with other features of the language. For example, numerous restrictions are needed on the kinds of parameters that can be used when explicitly re-defining one of the standard operations for a user-defined mode (e.g., the first formal parameter to an ASSIGN routine may not be bound by COPY [4, p. 164]).

d. It is extremely difficult to modify a language to meet a requirement as general as this. Making a language simple involves the deletion of numerous features. However, as will be discussed in Section XV of this chapter, there are very few features in CS-4 that are not needed to satisfy requirements in the Tinman.

6. Translator Restrictions (I6).

a. Degree of compliance: P

b. CS-4 partially satisfies this requirement, in that the language places a limit on the length of identifiers (viz., 32). However, CS-4 specifies neither the maximum number of array dimensions, the maximum level of parenthesis nesting, nor the maximum number of identifiers in programs.

c. It would be a minor modification to the language and to a well-structured implementation to specify language-defined limits for the number of array dimensions, level of parenthesis nesting, and the number of identifiers.

7. Object Machine Restrictions (I7).

a. Degree of compliance: T

b. Lacking restrictions of the kind described in I7, CS-4 completely satisfies this requirement.

## Section XI. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

1. Efficient Object Code (J1).

a. Degree of compliance: PT

b. CS-4 has a wide variety of features that promote the production of efficient object code, but there are also some facilities that may incur run-time overhead even when not used. The following sub-paragraphs illustrate both issues.

(1) Features facilitating efficiency.

(a) The mode parameters RANGE (for integers and reals), PRECISION (for reals and fractions), and string-size can be used by the compiler to determine the minimum number of bits that must be allocated for the representation of an object's value. These parameters are not specified in terms of bits, but in terms of the (integer or real) range of the value space, the decimal digits of precision, and the number of characters, respectively [4, p. 37, 44, 59, 100].

(b) The user has explicit control over whether procedures pass copies of actual parameter values or simply reference those values [4, p. 144]. This control is independent of whether the parameters are intended for input only, output only, or both input and output.

(c) Procedures declared to be OPEN will have their bodies expanded inline at each point of call, thereby trading off space for reduced invocation times. In all other ways, the semantics of OPEN and normal CLOSED procedures are identical [4, p. 148]. Procedures containing assembly or machine language code can also be declared OPEN and expanded inline.

(d) A NORECALL attribute can be associated with procedures. It is used by a compiler in optimizing multiple procedure calls having the same argument values [4, p. 150].

(e) Checking-directives are available to disable the generation of run-time checking code [4, p. 178].

(f) Many of the language-supplied procedures and operators will be invoked at compile-time if all of their arguments or operand values are known at compile-time. The resulting values can be used not only in contexts requiring compile-time values, but also in contexts that normally would require run-time invocations [4, p. 173].

(g) Although conversions are possible using explicit calls on construction routines, implicit conversions resulting in unknown overhead are not permitted [4, p. 26, 28].

(h) Storage allocation may be specified as either AUTOMATIC (within name scope levels), STATIC (at program level), SHARED (between processes at program level), or ABSOLUTE (for machine-dependent objects of mode MSTRUCTURE) [4, p. 20]. Pointers and HEAP storage requiring garbage collection are not provided.

(i) Recursive procedures are not permitted [4, p. 140]. Therefore, AUTOMATIC data (but only the dope vectors for dynamic arrays and strings) may be laid out statically, with restricted scopes of access, and not require references through a display. The language specification uses the terms "allocate" and "deallocate" to indicate when the data's storage is dynamically made available or unavailable, and where routines are invoked for initiation (of values for data and states for processes) and termination.

(j) Efficiency of storage using overlaid data can be achieved in three ways:

by compiler-determined storage layouts for AUTOMATIC data [4, p. 20];

by compiler-determined storage layouts for objects of discriminated UNION mode [4, p. 113]; and

by choosing storage-unit-values such that component fields in machine dependent MSTRUCTURES overlap [4, p. 183].

(k) The user can control the efficiency of initiate and terminate routines using the OPEN attribute as discussed above. In addition, object declarations using language supplied modes can specify that the initiate routine is not to be invoked [4, p. 18]. The terminate routines for the language-supplied modes perform no action [4, p. 26]; therefore, they can be implemented as body-less OPEN procedures that require no overhead.

(2) Features with hidden run-time expense.

(a) Array bounds need not be known at compile-time and thus require a "dope vector" in the general case. However, a dope vector will have to be used even when the bounds are known at compile-time (e.g., when the array is passed by reference as an argument to a routine whose formal parameter is an array with run-time determinable bounds).

(b) CS-4 has fairly powerful facilities in the area of exception handling and parallel processing. It is not obvious that run-time expense is avoided when these features are not used.

c. CS-4 comes very close to satisfying this requirement. In particular, the checking-directives mentioned above (b(1)(e)) give the programmer explicit control over efficiency versus reliability tradeoffs. Although there are a few features that may have hidden run-time expense (sub-paragraph (2) above), such features were designed to balance efficiency with reliability (possibly at the expense of simplicity), and could not be easily modified to completely satisfy this requirement.

2. Optimizations Do Not Change Program Effect (J2).

a. Degree of compliance: TU

b. This requirement is more a function of the translator than of the language itself, but it should be mentioned that there is nothing in CS-4 that prevents this requirement from being satisfied. For example, since the rules for expression evaluation result in side effects being carried out in left-to-right order, there is no danger of different translators producing object programs yielding different results for the same source program.

3. Machine Language Insertions (J3).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement almost completely via its MPROCEDURE direct code facility [4, Section 8.2]. The syntax and semantics of MPROCEDUREs (and their parameters) are consistent with those of normal procedures (e.g., they may be expanded inline). The rules are consistent, but not identical, because certain machine and implementation dependencies cannot be hidden: e.g., the machine-dependent register location of an actual parameter is specifiable, and may be used in the assembly code [4, p. 195]. A minor difference between CS-4 and J3 lies in the nature of the encapsulation of the machine language insertions. Instead of permitting direct code "only within the body of compile-time conditional statements" (as stated in J3), CS-4 employs MPROCEDURE...END brackets and requires the specification of a TARGET machine. Also, the direct code facility may only be used by programs possessing the MPROCEDURES authority [4, p. 194].

c. The modifications required to completely satisfy this requirement are dependent on adding compile-time conditional statements to the language (see I4).

4. Object Representation Specifications (J4).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement fairly well, via the MSTRUCTURE mode [4, Section 8.1]. With this feature, the user can specify the order and width of fields, the presence of "don't care" fields, and data alignment; it is also possible to associate source language data (but not programs) with special machine addresses. The use of MSTRUCTURES is restricted in a similar fashion to

MPROCEDUREs; the program must possess the MSTRUCTURES authority [4, p. 180]. One facility that CS-4 lacks in the area of object representation specifications is the ability to indicate a degree of packing for non-MSTRUCTURE data.

c. Modifying the language to allow a packing attribute on any data declaration is a relatively minor change. However, the effect on the implementation is non-trivial and can cause a great deal of run-time overhead for unpacking and implicit representational conversions. For example, it is non-trivial to implement by-reference parameter passing for a single Boolean variable in a packed array of Booleans.

5. Open and Closed Routine Calls (J5).

a. Degree of compliance: T

b. CS-4 provides OPEN and CLOSED as compilation class attributes for procedures [4, p. 148].

## Section XII. PROGRAM ENVIRONMENT

1. Operating System Not Required (K1).

   a. Degree of compliance: T

   b. None of the features in the CS-4 language itself [4] requires the existence of an operating system. CS-4 has an Operating System Interface [5] to make available to the language existing operating system capabilities in a standard fashion. Should a particular capability (or an entire operating system) not exist, they can be implemented directly in the OSI. It should be noted, however, that certain language features specifically required by the Tinman (e.g., parallel processing, G6) require run-time support (generally known as an operating system).

2. Program Assembly (K2).

   a. Degree of compliance: P

   b. The ACCESS mechanism of CS-4 partially satisfies this requirement.

3. Software Development Tools (K3).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

4. Translator Options (K4)

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

5. Assertions and Other Optional Specifications (K5).

   a. Degree of compliance: FU

   b. Although CS-4 provides no facilities (except comments) for assertions, etc., the language has nothing in it that would prohibit these forms. It should be noted, however, that interpretation of assertions, etc. (specified as implementation-dependent in K5) has a potential interaction with exception handling.

## Section XIII. TRANSLATORS

1. No Superset Implementations (L1).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

2. No Subset Implementations (L2).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

3. Low-Cost Translation (L3).

   a. Degree of compliance: P

   b. CS-4 partially satisfies this requirement. However, the size and complexity of the language may interfere with low-cost translation. Modifications to the language to meet this requirement are probably not desirable because conscious design trade-offs were made in the language in favor of such goals as reliability and efficient object code over fast translation.

4. Many Object Machines (L4).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

5. Self-hosting Not Required (L5).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

6. Translator Checking Required (L6).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

7. Diagnostic Messages (L7).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

8. Translator Internal Structure (L8).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

9. Self-Implementable Language (L9).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

## Section XIV. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only (M1).

a. Degree of compliance: P

b. All the language features in CS-4 are within the state-of-the-art with the possible exception of the data abstraction facilities. Although some existing languages (e.g., SIMULA-67) support data abstraction, this is an on-going research area. It should be noted, however, that such facilities are specifically required in E1 and E5.

2. Unambiguous Definition (M2).

a. Degree of compliance: P

b. Although CS-4's semantics have not been defined formally, an attempt has been made to be rigorous and complete. However, as has been noted under specific requirements, there are places where the language manual is incomplete, especially with regard to the Operating System Interface. On the whole, the manual is well-organized and quite readable, and although not quite suitable as a user introduction to the language, it does much better in this regard than most language reference manuals. Its main failing comes from the manner in which features are defined. Rather than explaining the capabilities of features as would be done in an introductory document, the reference manual frequently states descriptions in the form of restrictions.

c. A moderate amount of revision would be required to ensure that the language is defined completely and unambiguously.

3. Language Documentation Required (M3).

a. Degree of compliance: U

b. The CS-4 Language Reference Manual [4] is not intended as introductory-level user documentation, although in that regard it is better than many other language defining documents.

4. Control Agent Required (M4).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

5. Support Agent Required (M5).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

6. Library Standards and Support Required (M6).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

## Section XV. CONCLUSIONS REGARDING CS-4

1. Objectives of CS-4 and Tinman.

CS-4 comes extremely close to meeting all the objectives of the Tinman. CS-4 is a recently designed language which incorporates the latest advances in HOL technology as well as features that support current program design methodologies.

2. Summary of Major Areas of Conflict between CS-4 and Tinman.

a. Data and Types. There are several major conflicts between CS-4 and Tinman. CS-4 does not have a fixed-point data type. In addition, CS-4 does not permit user-defined character sets.

b. Operations. There are no really major conflicts but there are a number of minor ones.

c. Expressions and Parameters. There are several conflicts in this area. CS-4's parameter rules for modes are not consistent with parameter rules for procedures. CS-4 does not have precisely the four kinds of parameters listed in the Tinman. CS-4 does not have the generic procedure capability called for, nor does it allow a variable number of parameters to procedures.

d. Variables, Literals, and Constants. The major conflict in this area is that CS-4 does not have pointers. In addition, CS-4 does not permit subsequences of enumeration types.

e. Definition Facilities. The major conflict in this area is that CS-4 does not permit operator definitions. In addition, user-defined types are not indistinguishable from built-in types.

f. Scopes and Libraries. There are no major conflicts in this area.

g. Control Structures. The major conflict in this area is that CS-4 does not have recursive procedures. There are, in addition, numerous minor conflicts.

h. Syntax and Comment Conventions. The major conflict in this area is the clumsiness of the mode declaration syntax. Smaller conflicts include CS-4's ambiguous grammar, unreserved keywords, use of a larger character set than 64 character ASCII, and variances from uniform referent notation.

i. Defaults, Conditional Compilation, and Language Restrictions. The major conflicts in this area are CS-4's lack of a conditional compilation facility and the fact that the behavior of erroneous programs is often unspecified.

j. Efficient Object Representations and Machine Dependencies. There are no major conflicts in this area. Minor ones include the fact that some CS-4 features may incur run-time overhead even when not used, and the fact that packing can only be specified in MSTRUCTURES.

k. Program Environment. The major conflict in this area is that CS-4 does not have a special facility for including assertions in programs.

l. Translators. The major conflict in this area is that CS-4 has features which make low-cost translation difficult.

m. Language Definition, Standards and Control. The major conflicts in this area are the fact that CS-4 data abstraction facilities are probably not within the currently accepted state-of-the-art and the fact that there are places where the language is incompletely specified.

3. Unnecessary Features in CS-4.

The unnecessary features in CS-4 (in the sense that they are not needed to satisfy the Tinman requirements) are the arithmetic aggregate modes (COMPLEX, VECTOR, and MATRIX), keyword parameters, and the SHARED(UNPROTECTED) attribute. The arithmetic aggregate modes were built in for notational convenience but could be removed if operator extension were added to CS-4. Keyword parameters also provide notational convenience and we recommend their retention. The SHARED (UNPROTECTED) attribute is redundant -- i.e., it is equivalent to STATIC -- and should be removed.

## 4. Recommendations concerning CS-4.

On the basis of the evaluation conducted in this chapter, we conclude that CS-4 comes very close to meeting the Tinman, both in its goals and in the specific requirements. CS-4 can be modified, sometimes quite trivially and sometimes with more difficulty, to meet most of the Tinman's requirements. There are some places, however, (noted throughout this chapter and in Appendix III) where we feel that it is the Tinman which should be modified.

# CHAPTER 5

# JOVIAL (J73/I) EVALUATION

## Section I. LANGUAGE SUMMARY

1. Lexical Properties.

a. JOVIAL J73/I, called JOVIAL throughout this report, is a free-format language containing an explicit statement delimiter, ";". No special meaning is ascribed to end-of-line; in fact, all issues concerning it are ignored in [14].

b. The language is defined using 59 characters, all of which appear in the 64-character ASCII subset. Other characters (*) (i.e., any an implementation wishes to support) may appear internal to character constants and comments.

c. JOVIAL names consist of arbitrary length sequences of letters, digits and special symbols (prime and dollar-sign). Only the first 31 characters are significant. The first character of any name must be either a letter or $. The dollar sign may have an implementation-dependent meaning. The prime (') is included as a break character.

d. Comments may be of any length, are unaffected by end-of-line, and are delimited by quotation marks. A slight complication arises from the simultaneous use of quotation marks to delimit DEFINE strings (see compile-time facilities). To avoid ambiguity, comments may not appear within such strings. Except for the above restriction, comments may be placed between any two language symbols (lexical units).

---

(*) The colon is used in the JOVIAL definition, [14], to form multi-word non-terminals. Whenever colon-separated words appear within the body of this chapter, unless otherwise defined, they should be considered to denote a JOVIAL non-terminal.

2. Data Types.

The JOVIAL data types are categorized in Figure 4.

a. Scalars. The five JOVIAL scalar data types are floating point, signed and unsigned integers, and bit and character strings. The size for all scalar variables may optionally be specified, along with their declaration. If not so specified, strings default to a length of one and numeric variables to an implementation-dependent size. All size definitions, except for character strings, are made in terms of bits. A single character occupies a byte which is of implementation-dependent length.

(1) Behavioral properties common to all scalar data types. Assignment and relations are defined for all scalar data types. Implicit conversions will sometimes be made when necessary to force type compatibility. The rules of type matching permit conversions to occur in one direction in the type hierarachy -- character string (lowest), bit string, integer (signed and unsigned), floating point (highest). Explicit conversion routines between all scalar types are available.

The relational operators in JOVIAL are: =, <>, <=, >, >=. In comparisons between strings of different size the shorter operand will be padded on the right for character data and on the left for bit strings. Bit strings are padded with zeros. The character padding element is not specified.

(2) Behavioral properties specific to numeric types.

(a) Arithmetic operators. JOVIAL supports a full complement of arithmetic operators; +, -, /, \ (modulo), and ** (exponentiation). The resulting type of a numeric:formula is usually that of the hierarchically higher operand (e.g., floating point + integer yields a floating point result). In this context, signed integers are placed above unsigned ones. The only exception to this rule is that exponentiation involving only integers will frequently yield a floating point result.

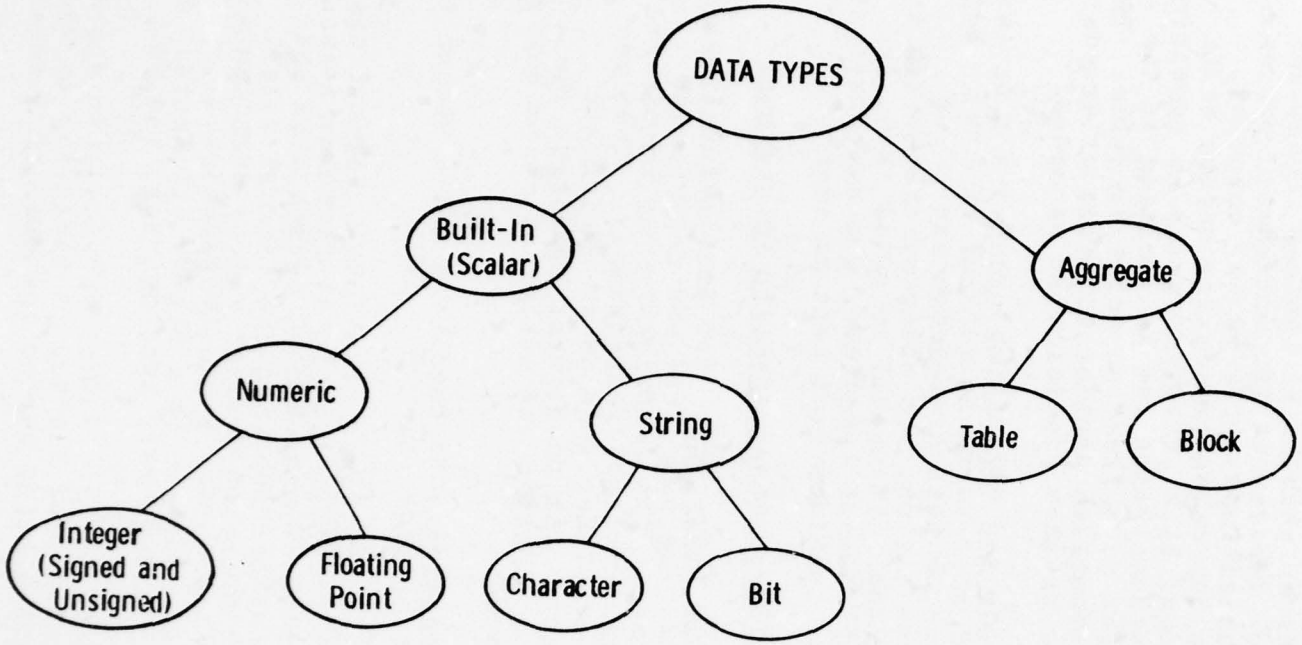


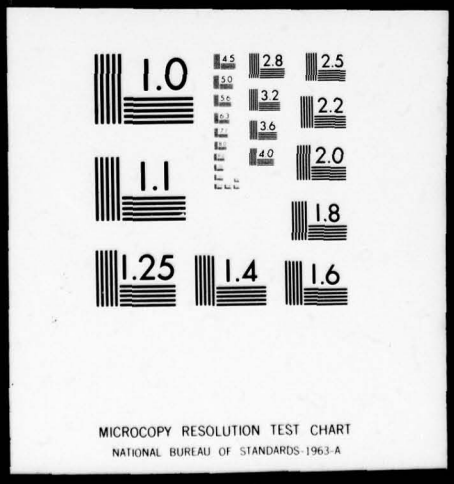

Figure 4. Data Types in JOVIAL

AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

15 of 28
ADA037634

15 OF 28
ADA037634



(b) Precision and scaling. Through size specifications the programmer can define the precision of numeric variables. Scale and precision management for arithmetic expressions is performed implicitly by the compiler. Overflow may cause truncation on the most significant bits without causing an exception. A truncation: option may be included in the definition of a floating point variable. This option specifies whether rounding or truncation is to occur when precision is exceeded.

(c) Precedence and evaluation order. A nine level operator precedence is defined in [14]. Beyond adherance to this precedence and parenthesization, the order of evaluation is unspecified. It should be noted that, in contrast to most languages, the exponentiation operator combines from left to right.

(3) Behavioral properties specific to strings.

(a) Substring operators. Two *pseudo-functions*, BIT and BYTE, are supported for referencing substrings. The functions apply to bit and character strings, respectively. They may appear on either side of an assignment statement. The programmer specifies the starting position and number of units being referenced.

(b) Concatenation. No concatenation operator is supplied in JOVIAL.

(c) Logical operations. The JOVIAL logical type is the bit string. Relational formulas return a bit string of unspecified length with the rightmost bit denoting the logical value. A one in this position denotes a true relation and a zero false. Five logical operators are provided which are applied on a bit-by-bit basis. These operators are: NOT, AND, OR, XOR, and EQV.

b. Aggregate Data Types. JOVIAL provides two aggregate data structures, tables and blocks. A table may be viewed as an array in which entries can be composed of several

scalar items. Blocks constitute a general structure in which any set of data objects (scalar or aggregate) may be grouped. The only operations supported for aggregate data types are component selection and parameter passing. Table declaration is quite complex, containing many different options. The programmer may define ordinary and specified tables. An explicit object representation is included in the declaration of a specified:table. Management of ordinary:table representation is done by the compiler with the programmer supplying packing density and allocation order. Packing densities are Non-packed, Medium packed, and Densely packed. The allocation order may be parallel or serial. Allocation order has meaning only for multi-item table entries. Serial allocation connotes that all words for a given entry appear consecutively in memory. In a parallel table all words for item 1 are grouped together, followed by those for item 2, etc. Specified tables permit the programmer to define the exact object representation of his data. Each item description of a specified table includes the word and bit position of that item in an entry. Item fields may overlap. The number of words per entry must be declared, but this is only used for indexing purposes and a given entry may exceed this number. In the case of variable length entries, the programmer is responsible for determining the proper index for locating a desired record.

3. Procedures.

JOVIAL supports both procedures and functions. A procedure may have any number of input and output parameters and is invoked by a call:statement. Functions may only contain input parameters, are invoked by the appearance of their name within a statement, and return a value of any scalar type.

a. Parameter Types. JOVIAL parameters are differentiated by the object type of the argument.

(1) Item parameters. A scalar data item may be declared as an input parameter, output parameter or both. Actual item input parameter values are copied into their associated formal parameters upon procedure entry. If an item is a formal output parameter, its value is copied into the proper actual parameter upon procedure exit.

(2) Table and block parameters. Aggregate data objects may only be declared as input parameters. They are passed by reference. No type-checking is done on these parameters. The associated memory area is treated as a bit-string and may be reorganized in any manner.

(3) Statement name parameters. A statement label may be declared as an input parameter. A GoTo within the subprogram body can cause a transfer of control to that label.

(4) Subprograms as parameters. Both functions and procedures may be passed as parameters. The attributes of such actual parameters must match those of the formal parameter declaration. The attributes include type and number of parameters and for functions the type of the result.

b. Recursion. JOVIAL does not directly support recursion. However, through the use of based procedures (see Storage Allocation) the programmer may explicitly simulate recursion.

4. Statements.

a. Null Statement. This permits extra semicolons to appear in programs.

b. Assignment Statement. The assignment statement in JOVIAL may have multiple targets. If several value destinations are specified, the assignments are made in left to right order. The type semantics of assignment were discussed in 2.a(1).

c. GoTo Statement. This statement allows control to be transferred to any label within an enclosing block or to any REFed label (see REF statement).

d. Return Statement. The RETURN statement when encountered causes procedure exit. An optional procedure name may be specified. The procedure so named must lexically enclose the RETURN and will be the one exited. Only the output parameters of the specified procedure will be assigned their proper values.

e. Stop Statement. Execution is terminated when this statement is reached.

f. Iterative Statements. Two forms of loop statements exist in JOVIAL.

(1) While statement. The WHILE loop executes a statement repeatedly until some logical condition becomes false.

(2) FOR Statement. A loop control variable must be included in the FOR statement. Clauses exist for initializing, incrementing and repeatedly replacing the value of the loop control variable. Termination is controlled by a clause identical to the WHILE statement.

g. IF Statement. The JOVIAL IF statement may have either one or two alternatives. Its syntax is somewhat unusual in that the word "then" does not appear (e.g., IF AA<Ø; AA=-AA;).

h. Switch Statement. This statement is equivalent to a numeric case statement. Depending on the integer value of a numeric:formula one of several labelled alternatives is selected. An optional default clause can be specified to be executed when no match is found. A command following a given case specifies that the succeeding case is also to be executed. The behavior when no match is found in the absence of a DEFAULT clause is undefined.

i. DEF Statement. A DEF statement promotes the availability of the names it includes to external modules. Any RESERVE variable (see Storage Allocation), statement, procedure, or function name may be DEFed.

j. REF Statement. The REF statement can extend the accessibility of any DEFed name to the scope of the REF.

5. Storage Allocation.

a. Variable Declaration. Three allocation methods are defined for variables, RESERVE, based and IN. RESERVE data is permanently allocated. Based variables require no storage memory. They are simply templates which may be laid upon any region of memory. Items declared to be IN are only

available within a given scope. Between entries to that scope their values become undefined. A subprogram may also be declared to be of one of the above types. The effect of such a declaration is that all variables declared within the procedure without an allocation:specifier are of the allocation type defined for the procedure. In addition, all other memory used by the routine (e.g., storage for a function result) is also allocated in the specified manner.

b. Static versus Dynamic Allocation. Since all item sizes and tables bounds are compile-time known and recursion is not permitted memory may be allocated statically in JOVIAL.

6. Process Scheduling.

JOVIAL contains no facilities for process scheduling.

7. Files and I/O.

[14] does not define any input or output facilities.

8. Exception Handling.

The only facilities existing in JOVIAL for exception handling are those the programmer implements through the use of statement name parameters. This mechanism is used to signal value changes caused by explicit conversion; however, no general facility exists for the purposes of exception handling.

9. Compile-Time Facilities.

JOVIAL contains fairly extensive compile-time facilities. Macros are supported by the DEFINE feature. Conditional compilation can be achieved by use of the !SKIP, !BEGIN, and !END directives. Compile-time variables are supplied. These include: BITSINBYTE, BITSINWORD, BYTESINWORD, and LOCSINWORD. The use of separate modules is encouraged by the !COPY directive (see [14, Section 6.1.1]), and REF and DEF statements (see 4.i and 4.j of this section). A compool facility is defined for the language.

## Section II. DATA AND TYPES

1. Typed Language (A1).

a. Degree of compliance: P

b. JOVIAL partially satisifies this requirement. JOVIAL requires that the type of each variable and each component of a composite data structure be explicitly specified in the source program. The language is not, however, strongly typed: on assignment or within expressions many required conversions are performed implicitly [14, Section 1.7.1.2]; the OVERLAY facility provides a free union allowing variables of one type to be used as if they had the characteristics of another type [14, Section 1.5]; subscripting a table to obtain an entry causes the resulting set of items to be interpreted as a bitstream [14, Section 4.11.2]; implicit conversions are performed when items are passed as parameters to procedures, but when tables or blocks are passed, neither implicit conversions nor type checking is performed [14, Section 2.2.3].

c. The modifications needed to satisfy this requirement are substantial. Adding strong typing to the language would drastically alter the nature and intent of the language. Type checking permeates so many features of the language (e.g., assignment, expressions, parameter passing) that attempting to add strong typing after the fact would require a major effort to ensure that all features affected have been dealt with. The modification required to the implementation would be major since type checking would have to be added throughout and implicit conversions removed.

2. Data Types (A2).

a. Degree of compliance: P

b. JOVIAL provides built-in data types for integer, floating-point, character string, and bit string. (Bit strings in JOVIAL are considered to be right adjusted -- when two bit strings of different lengths are involved in an expression, the shorter is padded on the left with zeroes [14, Section 1.7.2]. The JOVIAL character string on the other hand is left adjusted.) Objects having one of the built-in types are called items in JOVIAL. For aggregate data objects the language allows tables (arrays of entries,

where an entry is a set of items), and blocks (heterogeneous structures of items, tables, and other blocks). The main points of disagreement with the Tinman are the lack of a fixed point type and the fact that there is not an explicit character type, but rather character strings.

c. The scope of modications required is discussed under A4 (Fixed Point Numbers) and A5 (Character Data). As will be seen, the needed changes are extensive.

3. Precision (A3).

a. Degree of compliance: P

b. JOVIAL permits precision specification for individual variables [14, Section 2.1.3.2]. When two floating point variables in an expression have different precisions an implementation dependent padding of the lower precision value occurs [14, Section 1.7.2]. JOVIAL provides no mechanism for specifying global (to a scope) precision for floating point arithmetic.

c. The modifications required to the language and the implementation are relatively minor. A global precision declaration would have to be added to the language. The impact on the implementation would be a change to the declaration processor which enters information in the symbol table.

4. Fixed Point Numbers (A4).

a. Degree of compliance: F

b. JOVIAL does not offer fixed point numbers aside from integers. The integers are either signed or unsigned, and are treated as exact quantities with a step size of one.

c. The modifications required to add fixed point numbers to the language are substantial. A large design effort would be required both to design the fixed point features and to ascertain their interactions with other features such as the arithmetic operations. In addition, these modifications would interact with the modifications needed to add strong typing to the language (A1).

d. The impact on the implementation would be substantial, affecting all phases of compilation.

5. Character Data (A5).

a. Degree of compliance: F

b. Although JOVIAL provides the user with a mechanism for defining enumeration types, the character set is not defined as an enumeration type. [14] does not specify the order of characters; therefore the result of character relational expressions [14, Section 4.6.2] is entirely implementation dependent.

c. To modify JOVIAL to meet this requirement would be a major effort. To allow the user to define his own character set and also allow literals composed of strings of such characters is not desirable and not within the state of the art of language implementation. Even choosing a fixed character set (e.g., ASCII), it would be difficult to make characters behave like any other enumeration (STATUS is really just a way of assigning constant names to integers). Allowing characters to behave like STATUS values implies that fixed integer values must be assigned to character values and all integer operations will then be defined on characters (e.g., multiplication!). To prevent such a situation really requires the redesign of the STATUS mechanism.

6. Arrays (A6).

a. Degree of compliance: P

b. JOVIAL requires user specifications of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. JOVIAL requires both upper and lower bounds to be fixed at compile time. The JOVIAL enumeration type (STATUS constants) has an explicit user-defined mapping onto the integers [14, Section 2.4]. Array dimensions may be declared with default lower bounds (equal to zero if not specified) and upper bound explicitly specified [14, Section 2.1.5.3.1]. The value of a type declared by enumeration may be used for array reference -- as any integer value might be [14, Section 4.11.1] -- but may not be used in a table declaration as a dimension bound.

c. A small amount of language revision is required to add a dynamic upper array bound to the language. On the other hand, modifications to the implementation will be significant. First of all, dope vectors will be needed to access dynamic arrays. Secondly, since JOVIAL does not have recursive procedures, the implementation need not allocate storage dynamically at procedure entry. Consequently, adding dynamic arrays may require the addition of a dynamic storage allocator to an implementation.

7. Records (A7).

a. Degree of compliance: P

b. JOVIAL permits entries of specified (i.e., machine dependent) tables to have alternative structures. This is accomplished by specifying the starting bit location of an entry. No discrimination condition is required. JOVIAL also has an OVERLAY facility that allows variables of arbitrary type to be overlayed without any discriminating condition [14, Section 1.5.2], thus circumventing type checking. In addition JOVIAL has a based allocation capability that allows arbitrary templates to be laid over any area in memory.

c. Hierarchically structured data can be achieved through the use of BLOCKs [14, Section 2.1.6]. However, all the component names must be unique since there is no name qualification and the only available operation on blocks is parameter passing.

d. Major modifications would be required to redesign the data structuring mechanism to allow suitable hierarchical structuring, and alternative record structures with discrimination. This is the case because such facilities are different in philosophy from the current data structuring facilities in the language and because of the interaction of such facilities with such features as assignment, parameter passing, and type checking.

## Section III. OPERATIONS

1. Assignment and Reference (B1).

a. Degree of compliance: P

b. JOVIAL has no provision for encapsulated type definitions; the user has no means for defining assignment and access operations. The JOVIAL assignment statement can be used only for built-in types; it is not permitted to assign one table to another or one block to another. (This restriction is not explicit in [14], but it is implied by the syntax for assignment:statement [14, Section 3.3.1], since the right side must be a formula (thus of built-in type [14, Section 4.1]), and the rule which states that the value type of the formula must match the type of the variable(s) on the left side (non-built-in types never match under the rules of [14, Section 1.7]). Many conversions required to perform assignment are invoked implicitly [14, Section 1.7.1.2]. Reference to a table entry will retrieve the entry interpreted as a bit string. Since JOVIAL allows data overlaying, reference to a variable may retrieve a value different from the one most recently assigned.

c. In order to comply with this requirement (in the absence of encapsulated type definitions) assignment would have to be defined for tables and blocks. This, however, would require a substantial change to the language definition, as well as the implementation, since table and block declarations currently do not define new types. A fair amount of design is required to determine all the implications of treating tables and blocks as new types, and also to define the semantics of assignment.

d. Another change required is to remove the overlay facility so that data reference will always retrieve the last assigned value. This change is relatively minor. However, to fully ensure that data reference will always retrieve the last assigned value, based data (i.e., pointers) and implicit overlays in specified tables would also have to be eliminated, but such facilities are required by D6 and J4.

2. Equivalence (B2).

a. Degree of compliance: P

b. JOVIAL provides built-in operations which can be used to compare any two items. These operations cannot be used for comparison of tables or blocks, however. When the types of the data objects do not match, or their sizes differ, JOVIAL may implicitly convert one of the objects before comparing [14, Section 4.6.2].

c. For floating point identity to be specified within a precision, the user must explicitly specify it himself (e.g., ABS(A-B)<EPSILON rather than A=B).

d. Substantial revisions are required to satisfy this requirement. In order to define comparison for tables and blocks, they must first be redesigned as types of the language as discussed in B1. In addition, the implicit conversions need to be deleted so that objects of different types cannot be considered equivalent, and comparison for floating point needs to be redefined to include a precision specification.

3. Relationals (B3).

a. Degree of compliance: PT

b. JOVIAL defines relational operations for numeric data and for all types defined by enumeration [14, Section 5.2.2]. It is not possible to inhibit ordering, nor is it possible to define unordered sets.

c. The required modifications to the language are non-trivial. A new, unordered STATUS type, without mapping onto the integers, would have to be added to the language. Literals for this type would have to be differentiated from ordered STATUS literals.

d. The effect on the implementation would also be non-trivial. Determining the equivalence of two unordered STATUS types would require a great deal of checking. For example, the compiler would have to determine that the following three types are all equivalent: ("A", "B", "C"), ("B", "C", "A"), and ("A", "C", "B").

4. Arithmetic Operations (B4).

a. Degree of compliance: PT

b. The required operations are provided by JOVIAL, except that integer division generates an integer result. JOVIAL offers a modulo operator "\" [14, Section 5.2.2].

c. A fairly minor modification is required to add a division operation for integers which returns a real result.

5. Truncation and Rounding (B5).

a. Degree of compliance: P

b. JOVIAL partially satisfies this requirement, since truncation only occurs when the result of an operation is outside the range specifications of the program. As stated in [14, Section 1.7.2] with regard to bit strings and integers: "If the size of the value exceeds the maximum size permitted for its type by an implementation, its size is adjusted. ...Leading bits are truncated." However, this truncation is implicit (no exception condition is signalled). With regard to floating point, however, JOVIAL allows the user to specify whether truncation or rounding is to be carried out (on the least significant bits) with the truncation:option [14, Section 2.1.3.2].

c. In order to completely satisfy this requirement, truncation of integers would have to generate an exception condition. This, however, would require the addition of an exception handling facility to the language (see our discussion of G7).

6. Boolean Operations (B6).

a. Degree of compliance: PT

b. JOVIAL provides the built-in Boolean operations AND, OR, NOT, XOR, and EQV. AND and OR may be evaluated in short circuit mode (and some implementations do so evaluate them) but [14] does not specify that short circuit mode must be used.

c. Fairly minor modifications are required to add the NOR operator and to require short circuit mode.

7. Scalar Operations (B7).

a. Degree of compliance: F

b. JOVIAL does not permit scalar operations or assignment on conformable arrays (tables) or records (blocks).

c. The modifications required are substantial. As discussed in B1, tables and blocks would first have to be redesigned as types of the language. Then rules for type equivalence would have to be specified and the definitions of applicable operations would have to be extended.

8. Type Conversion (B8).

a. Degree of compliance: P

b. JOVIAL provides explicit conversion functions among integer, floating point, bit string, and character string data [14, Section 1.7]. JOVIAL does not provide conversion operations between the object representation of numbers and their representations as characters. It is important to note that many of the JOVIAL conversion operations are actually a different interpretation of an unchanged bit pattern (e.g., the conversion from floating point to character interprets a floating point number as "n" characters right-adjusted [14, Section 1.7.1.2]). JOVIAL provides implicit type conversions for type incompatibilities in expressions, assignments, and parameter passing [14, Sections 1.7.1.2, 2.2.3, and 3.3.2]; the built-in types are considered to be a hierarchy (float, integer, bit-string, character-string, in high to low order), and implicit conversions can be applied from one type to any type higher in the hierarchy. However, because the only conversions that change an internal representation are those from integer to floating point, very little run-time overhead is likely to be incurred.

c. As discussed in A1, implicit conversions permeate the language to such a large extent that to remove them would require a major effort and substantially change the nature of the language.

9. Changes in Numeric Representation (B9).

a. Degree of compliance: P

b. The only ranges permitted in JOVIAL are those determined from the number of bits declared for integer or float items. Explicit conversions are not required from one range to another. However, there is no run-time exception on truncation.

c. To comply with this requirement, a substantial modification would be needed to add an exception handling capability to the language (see our discussion in connection with G7).

10. I/O Operations (B10).

a. Degree of compliance: F

b. The JOVIAL language specification provides no operations allowing programs to interact with files, channels or devices. All such operations must be supplied in a procedure library; the definition of such a library is outside the scope of [14].

c. A large effort would be required to design a suitable set of I/O operations that fit cleanly into the language, provide the necessary capabilities, and are capable of interfacing with any existing operating system/file system without being implementation dependent.

11. Power Set Operations (B11).

a. Degree of compliance: P

b. Although JOVIAL does allow enumeration types, it does not offer power sets. It does offer bit strings and all the logical operators described under B6 which are sufficient to define union, intersection, difference and complement operations.

c. A moderate amount of work would be required to add power sets of enumeration types directly into the language. This would involve defining the form that power sets should take, checking for interactions with other features, especially enumeration types, and defining the appropriate operations on power sets.

## Section IV. EXPRESSIONS AND PARAMETERS

1. Side Effects (C1).

a. Degree of compliance: F

b. As stated in [14, Section 4.8.3], "the evaluation order of formulas is unspecified..." Thus, side effects need not occur in left-to-right order.

c. The change to the language to require left-to-right evaluation is very minor. A moderate amount of revision would be necessary to an implementation if it uses a different order of evaluation. This would also have a major effect on optimization (see our discussion in connection with J2).

2. Operand Structure (C2).

a. Degree of compliance: PT

b. JOVIAL has ten precedence levels, but some of these levels do not generally appear in a precedence table (e.g., indexing) [14, Section 4.8.3]. JOVIAL has some exceptions to standard combining rules: for example, the exponentiation operator, "**", combines left-to-right instead of right-to-left. Another uncommon feature of JOVIAL's expression definition is that two adjacent operators are allowed (e.g., A*-B is valid) [14, Section 4.8].

c. A fairly minor change would be required to make exponentiation right associative.

3. Expressions Permitted (C3).

a. Degree of compliance: T

b. JOVIAL satisfies this requirement. Variables but not constants are allowed as the targets in assignment statements [14, Section 3.3.1] and as the output arguments of procedure calls [14, Section 3.10.1]. Anywhere else a variable reference is allowed an expression is also allowed.

4. Constant Expressions (C4).

a. Degree of compliance: P

b. JOVIAL partially satisfies this requirement. There is a facility for using constant expressions (called numbers, which is not an especially intuitive term for this concept) in some contexts, such as preset lists and size:specifiers in item:descriptions [14, Section 2.1.3.1]. It is implicit that evaluation be at compile-time, at least in some cases.

c. However, there are many contexts in which constants are required and where literal values (not expressions) must be used. Examples of this situation are table bounds [14, Section 2.1.5.3] and switch:point:index:group [14, Section 3.8.1].

d. A moderate amount of revision would be required to the language to locate and change all the places where constants but not constant expressions are currently allowed.

e. The change to the implementation would be fairly minor, since the facility to evaluate constant expressions at compile time currently exists.

5. Consistent Parameter Rules (C5).

a. Degree of compliance: F

b. JOVIAL fails to satisfy this requirement, the main language deficiency being the inconsistencies in the rules for parameter passing to procedures [14, Section 2.2.3]. For example, tables and blocks may be input parameters but not output parameters; item parameters may be preset or overlaid but table and block formal parameters may not. (It might also be pointed out here that the parameter rules for procedures differ from those for "DEFINE"s, thus violating C5. Because of the different nature of these two language facilities, however, such variances are not surprising and probably not detrimental to language understandability.)

c. A very large effort would be required to satisfy this requirement. The parameter mechanism would have to be completely redesigned to make the rules consistent and this would substantially change the nature of the language.

6. Type Agreement in Parameters (C6).

a. Degree of compliance: P

b. JOVIAL partially satisfies this requirement. Type checking is carried out for item parameters, but implicit conversions may be invoked. Type checking is not carried out for table or block parameters [14, Section 2.2.3]: "The attributes of the formal and actual parameters should match. However, no such restriction is enforced. If the attributes do not match, the programmer is responsible for ensuring that the location(s) he wishes to use for the formal parameter are those he specified."

c. As has already been discussed under A1 and B8, deleting implicit conversions and imposing strong type checking would require a large effort.

7. Formal Parameter Kinds (C7).

a. Degree of compliance: P

b. JOVIAL distinguishes in the argument list and formal parameter list between input parameters and output parameters [14, Sections 2.2.1.1, 2.2.2.1, 3.10.1, and 4.10.1]. It should be noted, however, that nowhere in [14] is there a prohibition against assigning to an input parameter. This will have different effects depending on whether the parameter is an item or an aggregate: the former is passed by value, and the latter by reference.

c. Functions may only take input parameters [14, Section 4.10.1].

d. In addition to the data parameters provided, JOVIAL allows control parameters: statement labels, and procedure or function names [14, Section 2.2.1.1].

e. Modifying the procedure mechanism to correspond exactly to the four classes in this requirement would be a large effort. In addition to large revisions to provide the required form of constant and variable parameters (classes one and two), an exception handling control parameter mechanism (class three) would have to be designed.

8. Formal Parameters (C8).

a. Degree of compliance: F

b. JOVIAL fails to satisfy this requirement. The JOVIAL user must specify the types of all formal parameters (except for "DEFINE"s). JOVIAL does not support the kind of generic facility envisioned in C8.

c. A large effort would be required to meet this requirement. It should also be noted that the requirement is inconsistent since the macro capability implied by the phrase "instantiated at compile time by the characteristics of their actual parameters" does not provide a full "generic" capability.

9. Variable Numbers of Parameters (C9).

a. Degree of compliance: F

b. As stated in [ 14, Section 2.2.3]: "Actual parameters must match in position, number, and type with the called procedure's formal parameters..."

c. The modifications required to support this requirement would be non-trivial. The parameter passing mechanism would have to changed to meet the requirement, possibly by augmenting it with an implicit conversion from the variable size list of parameters on the calling side to a variable size array which is what the procedure itself would receive.

## Section V. VARIABLES, LITERALS, AND CONSTANTS

1. Constant Value Identifiers (D1).

a. Degree of compliance: P.

b. JOVIAL partially satisfies this requirement: the DEFINE facility can be used to associate constant values (for item data) with identifiers. However, this facility is a general macro replacement mechanism which works by string replacement independent of context; thus there is nothing to prevent the programmer from using DEFINEd names for constants in contexts where identifiers are not permitted. Worse still, unexpected results may occur; for example, if K is DEFINED as "2 + 3", then the result of K * 5 is not 25, but rather 17 ("2 + 3 * 5").

c. Even with the DEFINE facility, there is no way to obtain identifiers which reference constant aggregates (i.e., tables or blocks).

d. A moderate amount of work would be required to add constant identifiers to the language. This effort would involve designing the form of declaration as well as appropriate rules and restrictions. In addition, suitable checking would have to be added to the implementation to prevent their use in writable contexts.

2. Numeric Literals (D2).

a. Degree of compliance: P

b. JOVIAL provides a syntax and a consistent interpretation for constants of built-in data types [ 14, Section 5.3]. However, [ 14] nowhere specifies I/O or the internal representations of external data.

c. The main effort involved here really involves adding I/O to the language (see our discussion in connection with B10). Once this is done, only a minor effort is required to ensure that numeric program literals and numeric data are consistent.

3. Initial Values of Variables (D3).

a. Degree of compliance: P

b. The JOVIAL user may specify the initialization (called "presetting") of statically allocated data only (static data is called RESERVE) [14, Section 2.1.7.2]. Dynamically allocated data may not be initialized. There are no default initial values. The reference document [14] does not mention that a variable should be defined before being referenced. RESERVE data which are not preset are considered "undefined."

c. It would be a minor change to the language to allow initialization of variables. However, a moderate amount of revision would be required to the implementation to support it and also to check for the use of uninitialized variables. Checking for uninitialized variables, moreover, can potentially require a large amount of run-time overhead.

4. Numeric Range and Step Size (D4).

a. Degree of compliance: P

b. JOVIAL allows the user to specify the size, in bits, for integers (fixed point numbers are not in the language) and these integers may be signed or unsigned. The user may not restrict the range more narrowly than this size (e.g., with 4-bit unsigned integers it is not possible to restrict the range to 0..9). High order bits are truncated when overflow occurs, with no exception generated. With regard to floating point numbers, the user can specify the size of the exponent (as well as the mantissa) in bits.

c. A moderate amount of effort would be required to modify the language and the implementation to require the specification of ranges on all variables and to provide appropriate checking. These modifications would change the nature of the language.

5. Variable Types (D5).

a. Degree of compliance: P

b. JOVIAL contains a variety of restrictions on the structure of data. The only records which can be array

(table) components are those comprising items; it is not possible to have a block as an array constituent. The number of dimensions in a table is restricted to 7.

c. A large amount of redesign would be required to remove the restrictions from JOVIAL to make it comply with this requirement. One aspect of this would involve making tables and blocks types in the language (see our discussion in connection with B1).

6. Pointer Variables (D6).

a. Degree of compliance: P

b. JOVIAL provides a pointer facility which can be used to build shared or recursively organized data structures; viz., the based alternative for allocation:specifier [14, Section 1.4]. However, declaring a data object as based means only that its data description will be used as a template for the area of storage referenced by the address formula; there is no user-controlled dynamic storage allocation in JOVIAL.

c. It is required in D6 that pointers be restricted to reference some declared type of data. JOVIAL completely fails to meet this provision; any integer variable or formula may be used as a pointer [14, Sections 1.4.1, 4.11.4, 3.10.1].

d. A large effort would be required to design a suitably restrictive and reliable pointer mechanism into JOVIAL to satisfy this requirement.

## Section VI. DEFINITION FACILITIES

1. User Definitions Possible (E1).

a. Degree of compliance: P

b. Aside from its macro (DEFINE) facility, which is not directly relevant to E1, and its procedure and function mechanism, JOVIAL is quite weak in the area of definitional capabilities. Specifically, there is essentially no capacity for user-defined data types. (The status:list:name [14, Section 2.4.2] is the closest which JOVIAL comes to user-defined types, but this feature lacks the security generally associated with data types. For example, the relationship between status and integer values is explicit, and status constants are usable as integers and vice versa.) JOVIAL's data facility is that of "data structuring", as found also in FORTRAN, COBOL, TACPOL, and PL/I.

c. There is no facility in JOVIAL for defining new prefix or infix operators nor for extending the meaning of existing operators.

d. An extremely major design effort would be required to add abstract data definition and operator extension facilities to JOVIAL. Such facilities should be incorporated into a language right from the start, during its initial design. Adding such facilities after the fact would be extremely difficult, if not impossible, because of the pervasive effects of such facilities. Interactions of such facilities with type checking, assignment and parameter passing would require a large design effort.

2. Consistent Use of Types (E2).

a. Degree of compliance: F

b. JOVIAL does not provide defined types and therefore fails to meet this requirement.

c. Meeting this requirement is clearly contingent on adding user-defined types to the language. Even so, it is not yet within the state of the art in language design to make user-defined types indistinguishable from built-in types.

3. No Default Declarations (E3).

a. Degree of compliance: T

b. There is no provision for implicit declarations in JOVIAL.

4. Can Extend Existing Operators (E4).

a. Degree of compliance: F

b. JOVIAL provides neither type definition nor operator extension facilities.

c. A major design effort would be required to add an operator definition capability to the language as has already been discussed under E1.

5. Type Definitions (E5).

a. Degree of compliance: F

b. JOVIAL has no facilities for defining new types.

c. As already mentioned under E1, a major effort would be required to add type definitions to JOVIAL.

6. Data Defining Mechanisms (E6).

a. Degree of compliance: P

b. JOVIAL partially meets this requirement. Data may be defined by enumeration (STATUS values) and as Cartesian products, but not by discriminated union nor as the power set of an enumeration type. The definitions are processed entirely at compile time.

c. As has already been discussed under A7, major modifications would be required to add discriminated union to JOVIAL. As has been discussed under B11, moderate revisions would be required to add power sets.

7. No Free Union or Subset Types (E7).

a. Degree of compliance: P

b. The OVERLAY facility, based variables, and implicit overlays in specified tables all provide free union. JOVIAL provides no type subsetting.

c. The OVERLAY facility can be easily deleted from the language. Implicit overlays in specified tables are part of the machine-dependent object representation specifications required by J4, and based variables are part of the pointer mechanism required by D6.

8. Type Initialization (E8).

a. Degree of compliance: F

b. Lacking type definition, JOVIAL has no facilities for meeting this requirement.

c. Meeting this requirement is highly dependent on adding type definition facilities to the language, which is a major change (see our discussions of E1 and E5.).

## Section VII. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed (F1).

a. Degree of compliance: PT

b. JOVIAL satisfies this requirement fairly well. Data may be permanent, temporary, or based. Permanent (RESERVE) data have the behavior of "own" variables in ALGOL; their allocation scope exceeds their access scope. For temporary (IN) variables, the two scopes are identical. However, because of the absence of checks for based data, this facility allows unlimited access scope to data which may not be allocated.

c. The possible abuse of based data stems from the fact that no allocation scope actually exists for this form of variable. A based data structure simply defines a template which may be laid upon any region of memory and does not require allocation. Limitations on the range of overlaying are needed. These are difficult to define while still preserving based data's essential properties. However, these properties will be lost in any case if strong type-checking is required. One method of enforcing the rules of F1, then, would be to require the programmer to explicitly define the memory regions (variables) and possibly locations within these regions upon which the based data template may be applied. This would necessitate a moderate amount of language redesign and compiler modification. Some run-time checking would be required to assure that the base formula has the appropriate value.

2. Limiting Access Scope (F2).

a. Degree of compliance: P

b. JOVIAL partially satisfies this requirement. In addition to the scope rules mentioned in the preceding paragraph, the DEF and REF facility allows access to be limited both where an entity is defined and where it is used. However, there is no facility for renaming to prevent name conflicts.

c. A renaming facility must be added to the REF statement to bring JOVIAL into full compliance with F2. Only minor language modifications, which do not impact other

language features, are required to accomplish this. The additional compiler support needed would be small.

3. Compile Time Scope Determination (F3).

a. Degree of compliance: T

b. The scope of JOVIAL identifiers is wholly determined at compile-time [14, Section 1.3.3]. JOVIAL access scopes are lexically embedded with the most local definition applying when the same identifier appears at several levels.

4. Libraries Available (F4).

a. Degree of compliance: PT

b. JOVIAL has only a small library of specified intrinsic functions [14, Section 4.10.3]. The JOVIAL COMPOOL feature allows easy sharing of source language library modules and facilitates the devlopment of such libraries. However, [14] does not specify any application-oriented data or operation libraries.

c. The development of extensive application libraries is outside the scope of language definition. To the extent that such libraries can be encouraged by language design, through mechanisms which provide safe and easy accessibility, JOVIAL complies with this requirement (see our discussions in connection with F5 and F6). The one limitation on ease of access rests in JOVIAL's failure to provide renaming capabilities; however, support of such facilities can be simply added (see our comments concerning requirement F2).

5. Library Contents (F5).

a. Degree of compliance: T

b. JOVIAL offers the COMPOOL facility, which satisfies this requirement. The !LINKAGE directive [14, Section 6.2] allows access to procedures written in other languages.

6. Libraries and Compools Indistinguishable (F6).

a. Degree of compliance: T

b. The JOVIAL COMPOOL allows the definition of any JOVIAL data structure or routine to be included in a compile-time accessible form [14, Section 2.6.3.1]. The user can select any data objects or routines from the named COMPOOL. JOVIAL offers also a COPY directive, which causes external source text to be included in the to-be-compiled text. When refererencing a COMPOOL-defined identifier, the program obtains also all the attributes of that identifier [14, Section 2.6.4.2].

7. Standard Library Definitions (F7).

a. Degree of compliance: F

b. The JOVIAL specification [14] does not define interfaces to operating system capabilities, peripheral equipment, or any other machine dependent capabilities. Such capabilities must be supplied in a library but the contents and use of the library are implementation dependent and not specified in [14].

c. The most glaring deficiency in this area is that [14] does not define input or output operations. To correct this flaw alone would require extensive additions to the language definition. While all compilers must support I/O in some manner, a common description could be difficult to implement on a given machine. Modification of existing compilers to provide these new constructs would require a large effort (see our comments in connection with B10 for further discussion of I/O).

## Section VIII. CONTROL STRUCTURES

1. Kinds of Control Structrures (G1).

a. Degree of compliance: P

b. JOVIAL provides control structures for sequential, conditional and iterative control. Using "based procedures" [14, Section 1.4.2] it is possible to produce a limited recursive control mechanism. No control structures are available in JOVIAL for parallel processing (pseudo or real), exception handling or asynchronous interrupt handling. The control structures of JOVIAL are composable.

c. JOVIAL's deficiencies in this area cannot be easily removed. The addition of the required facilities would entail major language revisions. Compiler managed recursion requires new (dynamic) methods of memory allocation (see our discussion in connection with G5). Parallel processing necessitates provisions for task creation, termination, and synchronization. In addition, controlled access variables or program regions must be introduced. This would be complicated by the multiple reference methods supported by JOVIAL (see our comments regarding requirement G6). Compiler support of recursion and parallel processing needs to be extensive.

d. Exception and asynchronous interrupt handling, while somewhat simpler to define and implement, constitute far from trivial additions. Exception situations must be explicitly defined throughout [14]. In addition, general treatment of such instances requires a much more detailed model of the JOVIAL-machine interface.

2. The GoTo (G2).

a. Degree of compliance: P

b. JOVIAL provides a GoTo operation with very few restrictions on its use. Not only may a GoTo jump to any labelled statement in the most local scope of definition or in the enclosing program module [14, Section 3.4.1] but a procedure may also have statement label input parameters and GoTo those labels [14, Section 2.2.3]. JOVIAL does not have label variables or numeric labels. The language does prohibit branching into loops [14, Section 3.9.2], in that

the result of such an operation is "undefined." In addition, the REF declarator [14, Section 2.6.2], permits GoTo operations which greatly abuse scoping rules. For example, using such a declaration, a jump may be executed into the middle of a procedure which is outside the scope of the GoTo Statement, or from within a REFed routine into a textual ancestor (*) which has not been activated.

c. Restricting the scope of control transfers would be relatively simple. However, in the context of such restrictions, label parameters must be removed from the language. The major drawback of this is that in the absence of exception handling facilities (see our discussions regarding G1 and G7) these label parameters provide a reasonably clean method of passing error recovery information.

3. Conditional Control (G3).

a. Degree of compliance: PT

b. The JOVIAL IF-statement is not fully partitioned in that the ELSE clause is optional [14, Section 3.7.1]. JOVIAL's IF-statement has an unusual syntax: IF condition; statement-executed-if-true; [ELSE clause;]. The word THEN does not appear.

c. JOVIAL also offers a case statement (called a SWITCH [14, Section 3.8]). The selection condition is an integer expression, and the cases are labelled with the integer values which cause selection. There is a DEFAULT case, which is selected when no other case matches. The use of comma vs. blank to govern "fall-through" behavior is error-prone.

d. Requiring conditionals to be fully partitioned necessitates only small language and compiler modifications. The ELSE and DEFAULT clauses may no longer be considered optional. The null statement is already available for use in situations where no action is desired.

---

* A textual ancestor of a procedure, P, is considered to be any other procedure in which P has been embedded (nested).

4. Iterative Control (G4).

a. Degree of compliance: P

b. The iterative control structure of JOVIAL permits the termination condition to appear only at the beginning of the loop [14, Section 3.9.1]. JOVIAL does not impose unnecessary overhead for common special case termination conditions. The JOVIAL loop control variable is global to the loop; [14] does not specify any meaning for loop control variables after loop termination.

c. Since [14] does not provide a specific definition of the loop control variable's value upon termination, the addition of a statement to the effect that it is unavailable would be trivial. However, while this does preclude implementation-dependent interpretations, it is not quite the same as a local definition due to a greater allocation scope. Freedom of placement for the termination test requires the addition of a new control statement. A general form of loop-while should replace the JOVIAL while statement. This involves minor changes to the language definition and compiler design. Probably the greatest effort would be that of agreeing on a suitable form for this statement.

5. Routines (G5).

a. Degree of compliance: P

b. Recursion is not explicitly available in JOVIAL [14]. (A recursive:directive, with neither syntax nor semantics specified, is listed [14, Section 6.0]. This directive is ignored.) The JOVIAL user does have the possibility to use "based procedures" to implement recursive routines. As this approach is not supported directly by the language, it is error prone and clumsy to use.

c. Some form of compiler-supported recursion was, obviously, meant to be included in the full JOVIAL/J73 language. The level I subset, however, does not define what form this was meant to take. The most likely reason for this rests in a desire to support only those facilities which do not require dynamic memory llocation. J73/I does not contain any such facilities and for this reason implementation would be complicated by the addition of

recursion. Variables declared to be IN (allocated upon procedure entry) within recursive procedures need to be allocated from a memory pool (commonly called a heap). Routines must be included in the run-time system for maintenance of this memory pool and references to heap variables.

d. No change to the language definition, [14], is required to support recursive procedures; however, the meaning of the !RECURSIVE directive should be defined. The unrestricted nature of the JOVIAL GoTo may pose problems. If a statement label within a recursive procedure is made available via a REF declaration sticky situations can arise. Consider, for example, a recursive procedure P. P already has several existing activation records (i.e., it has been called but not exited several times). A GoTo causes transfer of control to a label internal to P from some external point. By the rules of [14, Section 2.6.2] no IN variables can be accessed and a RETURN would be illegal if the GoTo was reached by way of a procedure call, but legal if it was reached by another GoTo. Such rules are difficult to understand and enforce.

6. Parallel Processing (G6).

a. Degree of compliance: F

b. JOVIAL contains no facilities for meeting the parallel processing requirement.

c. The addition of parallel processing facilities to JOVIAL would constitute a major task. The effort would be more difficult than the complete definition of such facilities would have been if done along with initial language design. The reason for this is that interaction with existing language features greatly complicates the required changes. For example, one needs the capability to prohibit simultaneous access and update of a variable by two parallel processes. In the present language this is made very difficult by the existence of multiple methods for addressing a storage location (e.g., by name, by overlayed name, or by based formula).

7. Exception Handling (G7).

a. Degree ofcompliance: P

b. JOVIAL fails to satisfy this requirement. The issue of exceptions -- either hardware-oriented (e.g., overflow) or software-detected (e.g., index out-of-bounds) -- is not addressed very deeply in [14]. The explicit conversion functions allow as an exception-handling argument a statement label to which an unconditional branch is made when the conversion causes a value change [14, Section 1.7.2].

c. I/O, and therefore I/O exceptions, are not defined in [14].

d. In terms of language definition alone this requirement necessitates extensive modifications. The issues of error detection are for the most part ignored in [14]. All possible exception conditions must be identified and a mechanism for recovery must be introduced. The lack of defined I/O facilities greatly complicates this task (see B10 for problems associated with their introduction). The difficulties encountered in implementation will depend upon the sophistication of the added facilities and the machine on which the programs are to run. Even with minimal facilities and a high degree of machine support this would require quite large additions to the compiler.

8. Synchronization and Real Time (G8).

a. Degree of compliance: P

b. JOVIAL provides no source language features for synchronization of parallel processes, nor for access to hardware interrupts or real time clocks.

c. Let us first mention that this requirement assumes that both requirements G6 and G7 have been met. The fact that JOVIAL fails to comply with either of these implies that a great effort is needed before G8 can even be considered. Given compliance with both G6 and G7, the facilities suggested here still constitute a major change to the language. The ability to specify relative priorities and permit delay on any control path necessitate sophisticated sequencing of parallel processes. To handle

asynchronous hardware interrupts one must be able to save the state of the present process and resume it at some future time.

## Section IX. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics (H1).

a. Degree of compliance: P

b. The JOVIAL language is free format with an explicit statement delimiter, allows the use of mnemonically significant identifiers, has a relatively easily parsed grammar, and does not permit abbreviation of identifiers or keywords. Although JOVIAL keywords may not be abbreviated some of the mnemonics are already so abbreviated that they are not easily read. The single character mnemonics used in declarations (F=float, U=unsigned integer, D=densely packed) are examples. Also, some mnemonics are poorly chosen (e.g. IN does not specify integer type but rather automatic allocation of storage upon entry (in?) to a scope). The absence of the word THEN in the IF-THEN-ELSE construct is unusual. In addition, table declarations contain many context-dependent symbols and values which can obscure readability. For example, the following is a valid specified table declaration:

```
TABLE T1, [3] P 5 F 5, 3 D [2,1]=[Ø] 2 (5);
```

c. It should also be noted that the JOVIAL grammar given in [14] is ambiguous, as revealed in the rule in [14, Section 4.8.1]:

```
numeric:formula ::=
    numeric:formula arithmetic:operator numeric:formula
```

The ambiguity is resolved in the rules for operator precedence.

d. As can be seen from the above example, JOVIAL's declarations can be hieroglyphic to those not intimately familiar with the language. The introduction of more significant and readable declarators and possibly guidewords would greatly ease the task of understanding JOVIAL programs. This necessitates no change in language semantics. The enhanced syntax would require only minor modifications to the scanning and possibly parsing modules of the compiler.

e. The ambiguity of JOVIAL's grammar can be justified as a means of facilitating readability of the defining document. Implementors are not restricted to use of the grammar in its specified for and in fact would probably wish to avoid the specified ambiguities. This could, however, lead to somewhat implementation-dependent syntax unless great care is exercised. We suggest that where it is felt necessary to use ambiguous productions, an unambiguous set of rules should also be included for implementation purposes. This constitutes a fairly minor addition to [14].

2. No Syntax Extensions (H2).

a. Degree of compliance: T.

b. JOVIAL has no facilities for modifying the source language syntax (the DEFINE feature is not regarded as violating this requirement).

3. Source Character Set (H3)

a. Degree of compliance: T

b. The JOVIAL source character set consists of the upper case letters A-Z, the digits 0-9, and the following special characters [14, Section 5.1]:
blank + - * / \ < > = : , ; () [ ] ' " ! $
All of these characters are in the 64-character ASCII subset.

4. Identifiers and Literals (H4).

a. Degree of compliance: P

b. The JOVIAL language definition provides formation rules for identifiers [14, Section 1.3.1] and literals [14, Section 5.3.3.1]. The prime (i.e., apostrophe) is used for an explicit break character in identifiers [14, Section 1.3.1]. No break character is defined for use within literals. End of line has no role in JOVIAL.

c. Full compliance with H4 can be reached with only minor changes to the language and scanning module of the compiler. One must permit a break character within literals (e.g., the apostrophe) and require separate quoting of multi-line character literals.

5. Lexical Units and Lines (H5).

a. Degree of compliance: P

b. JOVIAL nowhere specifies any function for end-of-line. Any implementation-defined "other characters" are accepted in character string constants and comments [14, Section 5.1].

c. The requirements of H5 can be met with minor additions to [14]. An explicit end-of-line constant for use within character literals must be defined. A prohibition of the continuation of a lexical unit across lines must be added. Both of these involve trivial changes to the compiler and do not impact other language features.

6. Keywords (H6).

a. Degree of compliance: P

b. JOVIAL's keywords are reserved and are 68 in number [14, Section 5.2.1]. These keywords are generally informative, although DEF, DEFINE, and DEFAULT are not completely clear. Some keywords, particularly the implementation constants describing the target machine, are intended to appear anywhere an integer constant or identifier can appear.

c. The freedom with which implementation constants can be used cannot be restricted without limiting their value. Possibly their definition as keywords is ill-advised, but allowing programmers to declare variables with these names would only lead to obscurity and confusion. The unclear keywords can be simply changed to more informative ones (e.g., DEFAULT which connotes a switch path to be taken when all others fail might be replaced by OTHERWISE). This would constitute a trivial change to the compiler's scanner.

7. Comment Convention (H7).

a. Degree of compliance: PT

b. JOVIAL comments are enclosed in quotation marks [14, Section 5.4] and may contain any character except a quotation mark. End-of-line has no effect on comments. JOVIAL comments may appear almost anywhere in programs. An

important exception is in connection with macros (DEFINE): since the define:string is delimited by quotation marks, comments may not appear between the define:name and the define:string. This, however, is a good place for the user to provide a comment explaining the define:declaration.

c. The JOVIAL comment convention may be trivially modified to accept end-of-line as a terminator. However, this implies that it could not be used to enclose a program region of arbitrary length and may be construed as a violation of the balanced parentheses requirement. The conflict between comment and DEFINE notations can be removed by defining a different comment delimiter. Use of this facility (i.e., commenting within DEFINEs) could lead to subtle errors due to the macro replacement of a DEFINEd name in a place where comments are not permitted. All in all, full compliance may be reached with great ease.

8. Unmatched Parenthese (H8).

a. Degree of compliance: T

b. JOVIAL has parentheses "()", brackets "[ ]" and BEGIN END as "parentheses". These must all be matched.

9. Uniform Referent Notation (H9).

a. Degree of compliance: F

b. The argument list for a function call is enclosed in parentheses "()" [14, Section 4.10.1]; the subscript list for a table reference is enclosed in brackets "[ ]" [14, Section 4.11.1].

c. JOVIAL uses pseudo-functions for its bit- and character substring operations [14, Section 4.11.2]: these pseudo-functions may appear on the left of an assignment statement.

d. Within the program body, no real difficulty arises from permitting parentheses to be used for both function and table references; however, ambiguities will be encountered if this is allowed within table declarations. A,[2]B and A,(2)B both have unique and valid meaning within a specified:table:declaration preset:part. If we limit our requirement to uniform reference within the program body

then compliance will necessitate a moderate amount of effort, otherwise a large amount is required.

10. Consistency of Meaning (H10).

a. Degree of compliance: P

b. JOVIAL has several inconsistencies: the equal sign "=" may be the assignment operator or a relational operator (e.g., IF XX=1; XX=2; ELSE XX=5; [14, Section 3.7.3]), the letter "M" in a table declaration specifies a medium packing density [14, Section 2.1.5.5] and in a floating point constant literal specifies the mantissa precision [14, Section 5.3.3]. The representation of status constants (the JOVIAL enumeration type) is V(name) which looks like a function call but is in fact an integer constant [14, Section 5.3.3.1]. The single quote may be used in identifiers as a break character [14, Section 1.3] but is also used to delimit bit and character string constants [14, Sections 5.3.4 and 5.3.5]. As mentioned in the preceding paragraph, the body of a define:declaration is delimited by double quotes, which are also used to delimit comments. The parameters in a define:declaration are preceded in the define:string by an exclamation mark, which also precedes all compiler directives [14, Section 6].

c. The overall notational change needed would require a moderate amount of language redesign. The modifications are all simple syntactic changes and therefore would require only small revisions to the compiler.

## Section X. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic (I1).

a. Degree of compliance: P

b. The behavior of any program in the event of run-time exceptions, compile-time errors or situations yielding undefined results (e.g., [14, Section 3.9.2]) are not specified and thus are entirely implementation dependent. Within a program all attributes must be declared or a JOVIAL specified default is selected (e.g., precision for floating point variables).

c. The task of eradicating all defaults from JOVIAL would not be easy. Many implementation-dependent defaults exist and permeate the entire language. Error situations are almost totally ignored in [14]. The definition provides the syntax and semantics of valid JOVIAL statements but supplies little information about actions to be taken when encountering invalid code. No error or warning messages are defined. The needed modifications to the defining document are extensive. Such changes would at least require moderate revision of the compiler.

2. Object Representation Specifications Optional (I2).

a. Degree of compliance: P

b. JOVIAL provides a very high degree of user control over data representation, but there are well-defined defaults when the user does not care. There is no user control over program characteristics such as open vs. closed subroutines, and reentrant vs. non-reentrant code generation.

c. Control over program characteristics such as open vs. closed routines and reentrant vs. non-reentrant code generation can be provided through the addition of new directives to the language. These additions could be easily made without seriously impacting any other language features. The needed modifications to the compiler would entail a moderate amount of work. Compilation of reentrant code may involve penalties in run-time efficiency.

3. Compile Time Variables (I3).

a. Degree of compliance: T

b. JOVIAL provides a set of compile-time constants, called machine:parameters [14, Section 1.2], which specify characteristics of the object machine. In addition, the DEFINE facility, COMPOOL, and certain compiler directives (such as !SKIP) provide a flexible compile-time capability to the user.

4. Conditional Compilation (I4).

a. Degree of compliance: T

b. JOVIAL satisfies this requirement, via its skip:directive [14, Section 6.1.2]. However, it would have been a somewhat more unified approach if the conditional compilation facility were part of the DEFINE mechanism.

5. Simple Base Language (I5).

a. Degree of compliance: P

b. JOVIAL satisfies this requirement fairly well. Aside from the DEFINE feature, JOVIAL lacks extension facilities; thus the entire language may be considered as the base. Complexities in the language include a somewhat baroque switch:statement and interactions between features such as storage allocation, presetting, and procedure parameters.

c. The extent of modifications required to achieve full compliance with this requirement greatly depends upon the additions made in conjunction with other Tinman requirements. For example, if class structures as discussed in section E are desired, some facility must be included within the "base" for their definition.

d. As the language stands, the modifications necessary for the definition of a simple base are relatively minor. A few features, such as the switch, could be removed and defined in terms of other constructs, but the result would not differ greatly from the total language.

6. Translator Restrictions (I6).

a. Degree of compliance: P

b. JOVIAL limits the number of array dimensions [14, Section 2.1.5.3.2] and the length of identifiers [14, Section 1.3.2]. Neither the number of nested parentheses levels in an expression nor the number of identifiers are restricted by [14].

c. Setting values for the various translator-dependent limits would not be a difficult task. The real problems rest in differentiating limits imposed by translator design and those inherent in the object machine. For example, if we limit the number of identifiers to 400, which seems to be in the range suggested, a translator on a 16-bit machine would require 6200 words to store all potential identifiers. This number is arrived at by assuming 2 character/word storage and noting that the first 31 characters of a JOVIAL name are significant. On a small machine this could be prohibitive.

7. Object Machine Restrictions (I7).

a. Degree of compliance: PT

b. No restrictions dependent only on the object environment are built into the JOVIAL language definition [14]. Nowhere in [14] is it stated that JOVIAL will report when a program exceeds the resources or capabilities of the intended object machine.

c. Reporting of resource overruns should be done by any "good" compiler and the inclusion of such a requirement within [14] is trivial. We should note that such inclusion runs counter to the nature of [14], in that in general error situations are not addressed.

## Section XI. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

1. Efficient Object Code (J1).

a. Degree of compliance: PT

b. JOVIAL satisfies this requirement fairly well. The data structures offered by the language are machine-oriented (e.g., size and precision are specified in terms of numbers of bits, table entries are regarded as bit strings), so that the programmer is always aware of the match between data and hardware. Array bounds are fixed at compile-time, avoiding the need for dope-vectors. Recursion is possible only at great pains by the programmer; when recursion is achieved, the costs are explicit. Data overlaying is permitted (there is no discriminated union). The programmer can specify the layout of tables as parallel or serial [14, Section 2.1.5.4], and can indicate the density of packing. Optimization directives such as !INTERFERENCE and !REDUCIBLE, are available [14, Sections 6.4 and 6.5].

c. An example of a language feature which results in possibly-unexpected overhead is implicit conversion. A misguided efficiency is also illustrated by this feature: the conversion from floating-point to character-string simply reinterprets the bits comprising the former as though they were characters. Although this results in no overhead, the behavior is not likely to be the one desired.

d. Removal of the implicit conversions existing in JOVIAL is discussed in B8. As can be seen from that discussion the changes are far from trivial.

2. Optimizations Do Not Change Program Effect (J2).

a. Degree of compliance: F

b. JOVIAL does not satisfy this requirement. The order of evaluation of formula components [14, Section 4.8.3] and of arguments to a routine [14, Section 2.2.3] is left unspecified. Thus side effects need not occur in left-to-right order.

c. This requirement has the effect of greatly limiting the number of optimizations applicable. Recognition of all

conceiveable side-effects is frequently difficult and sometimes impossible. The JOVIAL compiler is permitted great freedom in terms of optimizing the user's program. Data structures may be rearranged and formulas evaluated in any order. Compliance with J2 would entail extensive compiler improvement or a potentially large penalty in run-time efficiency. These effects would be limited, however, to the optimization section of the compiler. The addition of this requirement to the language definition [14], however, would be trivial.

3. Machine Language Insertions (J3).

a. Degree of compliance: P

b. JOVIAL makes no provision for the insertion of machine language code. However, the user has access to machine capabilities at a very detailed level in the data representation. An OVERLAY specification can define an absolute memory address for the storage of data. By using JOVIAL's "specified table" capability, the programmer can associate an identifier with fields of any width within a word or words. The OVERLAY facility allows the same field to be used, for instance, as an unsigned integer and as a bit string. Thus, although these fields may be manipulated only through JOVIAL constructs, the effect is to allow quite detailed control of the hardware on certain types of computers.

c. The REF mechanism and !LINKAGE directive can be used to provide a machine language insertion capability. This directive is, of course, implementation-dependent. The encapsulation of these routines can be achieved via the conditional compilation of the associated REF.

d. One stumbling block still remains, however. Since JOVIAL provides no mechanism for defining open (in-line) procedures every reference to a machine language insertion necessitates a procedure call. This can be disastrous when these insertions are being used to achieve improved efficiency. The introduction of open procedures is discussed in J5 but further work might be required to support routines called through a !LINKAGE directive.

4. Object Representation Specifications (J4).

a. Degree of compliance: P

b. JOVIAL offers comprehensive capabilities for the specification of data representations, as described above in conjunction with J1. However, the facilities provided are not encapsulated: they are specified together with the logical description of the data.

c. Separation of the logical definition and object representation description would be difficult in JOVIAL. The specified:table:declaration is basically an ordinary:table:declaration with added facilities for specifying item placement. A great amount of redundancy would be required for separate specification, and compilation would most likely become more difficult. The !SKIP, !BEGIN and !END directives allow any specified:table:declaration to be encapsulated in compile-time conditional statements.

5. Open and Closed Routine Calls (J5).

a. Degree of compliance: P

b. JOVIAL contains no facilities for specifying open vs. closed routine calls.

c. From a language definition viewpoint this requires only a trivial change. The introduction of an !OPEN directive would suffice. Implemenmtation, however, would not be so simple. Several features of JOVIAL would cause complications. For example, the meaning of a REFed statement label would be ambiguous if it appeared within an open procedure with several calls. In addition, the meaning of an open based procedure is not immediately obvious. The DSIZE function [14, Section 4.10.11] would have to be extended. In summary, for a coherent treatment of open and closed routine calls a moderate amount of language redesign and compiler revision would be needed.

## Section XII. PROGRAM ENVIRONMENT

1. Operating System Not Required (K1).

a. Degree of compliance: T

b. The JOVIAL defining document, [14], makes no reference to the existence of an operating system. Language features, common to other languages, which rely on some OS support are not defined in JOVIAL. These include input and output, parallel processing and interrupt capabilities.

2. Program Assembly (K2).

a. Degree of compliance: T

b. JOVIAL encourages modularity through the !COPY and !COMPOOL directives and the DEF and REF statements.

3. Software Development Tools (K3).

a. Degree of compliance: U

b. The existence of support tools is not discussed in [14]. It should be noted that some of the recommended tools do exist for at least one implementation of the language.

4. Translator Options (K4).

a. Degree of compliance: U

b. Several options such as listing suppression are supported in JOVIAL through directives, but general compiler options are not discussed.

5. Assertions and Other Optional Specifications (K5).

a. Degree of compliance: F

b. Except for the standard comment mechanism JOVIAL does not provide any facility for the inclusion of assertions, assumptions, axiomatic definitions or units of measure.

c. The introduction of additional commenting forms for these purposes is straightforward.

## Section XIII. TRANSLATORS.

1. No Superset Implementations (L1).

   a. Degree of compliance: F

   b. JOVIAL J73/I is a subset language and therefore totally fails to meet this requirement. Several keywords and directives are defined simply for compatibility with the full language.

   c. Removing the keywords and directives which are not defined in J73/I would be trivial. Outlawing superset implementations, however, is antithetical to the intentions of its designers for the language to be a subset of full J73.

2. No Subset Implementations (L2).

   a. Degree of compliance: U

   b. [14] does not consider the possibility of subset implementations. It would be trivial to add such a prohibition to the defining document.

3. Low-Cost Translation (L3).

   a. Degree of compliance: U

   b. The efficiency of compilation is outside the scope of language definition. To the extent the language design affects compiler speed, JOVIAL meets the requirement. No features of the language require excessive computation to determine their meaning.

4. Many Object Machines (L4).

   a. Degree of compliance: U

   b. This requirement applies only to language implementation.

5. Self-Hosting Not Required (L5).

   a. Degree of compliance: U

b. Cross-compilation is an issue of language implementation, not design.

6. Translator Checking Required (L6).

a. Degree of compliance: U

b. Compiler responsibilities are not discussed in [14].

7. Diagnostic Messages (L7).

a. Degree of compliance: F

b. [14] does not suggest a set of error and warning messages.

c. It would require a large effort to define a set of error and warning messages for JOVIAL. The major difficulty arises from the unclear nature of many possible error situations in [14]. Frequently, potential misuse of JOVIAL contsructs is ignored (see our discussion regarding M2).

8. Translator Internal Structure (L8).

a. Degree of compliance: T

b. [14] does not dictate the internal structure of a JOVIAL compiler.

9. Self-Implementable Language (L9).

a. Degree of compliance: T

b. The DAIS JOVIAL J73/I compiler is written in JOVIAL.

## Section XIV. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only (M1).

a. Degree of compliance: T

b. No features of JOVIAL are beyond the current state-of-the-art.

2. Unambiguous Definition (M2).

a. Degree of compliance: F

b. The omissions and unclear specifications of the defining document, [14], are numerous. The semantics given for many language features are vague and open to several interpretations (see c below). Frequently the semantics of syntactically valid constructs are omitted or incomplete (see d, e and f below). Examples which are intended to illuminate sometimes achieve the opposite effect by being incorrect (see g and h below). Use of such words as "can", "may" and "should", without further explanation only lead to confusion (see i below). Occasionally, the consequences of the stated semantics seem to be unintended (see j below).

c. Type matching rules, one of the most importnt aspects of any language definition, are unclear for JOVIAL. The following are the first two sentences of [14, Section 1.7] pertaining to these rules, "Two values are said to match if their value types and sizes are identical. In contexts requiring a particular value type or an equivalence of value types of two operands, the value type of a value can be changed...". The document then goes on to describe implicit conversions. The question still remains, however, "What is the meaning of 'types must match'?" This is of crucial importance since the requirement about 'matching' types appears throughout [14].

d. The semantics of a FOR statement are incomplete. It is obvious from the syntax that the initial:value clause for a control:variable may be omitted; however the semantics of such an omission are not specified [14, Section 3.9].

e. In the description of the RETURN statement [14, Section 3.6.2], it is stated, "If a return is from a (lexically) outer procedure, any output parameters of the

inner procedure that contain the return:statement are not set." But what happens if the output parameter of an inner procedure happens to be associated with an output parameter of the exited procedure?

f. Probably the most glaring omission of [14] occurs in the definition of string padding [14, Section 4.6.2]. Bit strings are specified to be padded with zeros, but the character padding element is never defined.

g. The Shell sort on [14, Page 3-12], at best, sorts in reverse order.

h. The result of the logical formula given on [14, Page 4-3], is said to be 3B'16571' when, in fact, it should be 3B'16471'.

i. In [14, Section 1.7.1.2] the following statement is made, "If the type of two values is B, U, S or F, a matching function can be invoked implicitly..." Does the word "can" imply that these implicit conversions are implementation-dependent? If so, it should be stated explicitly.

j. According to the language semantics given in [14], the following statement is illegal if II is an integer (even unsigned):

```
II=II**II;
```

The reason it is illegal is that an exponentiation where the exponent is not a positive integer constant returns a floating point value [14, Section 4.8.2] and such a value may not be implicitly converted to integer type [14, Section 1.7].

k. It would be quite difficult to remove all ambiguous specifications from [14]. Large and numerous sections of the document would have to be rewritten.

3. Language Documentation Required (M3).

a. Degree of compliance: F

b. The language definition includes the syntax, semantics, and examples of each language construct. However, the syntax is frequently ambiguous (see our discussion in connection with M1) and the semantics unclear

(see our comments regarding M2). In addition, examples are sometimes incorrect (see our remarks concerning requirement M2).

c. The scope of modifications required to produce a clear and unambiguous language definition have already been seen to be substantial (see our final remark regarding M2).

4. Control Agent Required (M4).

a. Degree of compliance: U

b. The control of language and compiler standardization is outside the scope of language definition.

5. Support Agent Required (M5).

a. Degree of compliance: U

b. Identification of "support agent(s) responsible for maintaining the translators and for associated design, development, debugging and maintenance aids" is beyond the scope of language definition. One should note that in reference to JOVIAL this identification process should be facilitated by the fact that support agents already exist.

6. Library Standards and Support Required (M6).

a. Degree of compliance: U

b. This requirement is beyond the scope of language definition. The comments of M5 apply here.

## Section XV. CONCLUSIONS REGARDING JOVIAL (J73/I)

### 1. Objectives of JOVIAL and Tinman.

The only goal of the Tinman that JOVIAL comes close to meeting is that of efficiency; JOVIAL does especially poorly with regard to reliability and readability. Even with regard to transportability, JOVIAL does not do very well since the language permits machine and implementation dependencies to permeate an entire program.

### 2. Summary of Major Areas of Conflict between JOVIAL and Tinman.

a. Data and Types. There are several major conflicts here. JOVIAL lacks strong typing and performs implicit conversions. In addition, it does not have a fixed point data type, user-defined character sets, dynamic arrays, nor discriminated variant records.

b. Operations. There are several major conflicts in this area. JOVIAL does not have assignment or comparison operations for tables or blocks, lacks I/O facilities entirely, and does not have a powerset type.

c. Expressions and Parameters. One conflict in this area is that JOVIAL does not require left-to-right evaluation. With regard to parameters, the rules are not consistent, implicit conversions are sometimes performed, the kinds of parameters do not correspond to those in the Tinman, there is no generic capability, and a variable number of parameters is not allowed.

d. Variables, Literals, and Constants. The major conflict in this area is the unreliable nature of the pointer mechanism. In addition, ranges cannot be specified on numeric variables, not all variables can be initialized, and there are various restrictions on the structure of data.

e. Definition Facilities. The major conflict is the lack of a type definition facility and an operator extension facility. In addition, there is no discriminated union or power set type in the language, and free unions (i.e., overlays) are permitted.

f. Scopes and Libraries. The major conflict in this area is the lack of a machine independent interface to operating system and I/O capabilities.

g. Control Structures. The major conflicts in this area are the lack of recursion, parallel processing and exception handling capabilities. In addition the language has an unrestricted GoTo statement.

h. Syntax and Comment Conventions. The conflicts in this area are the ambiguous grammar, the poor choice of keywords, and the lack of uniform referent notation.

i. Defaults, Conditional Compilation, and Language Restrictions. The major conflict in this area is the fact that often the behavior of a construct, particularly in the case of an error situation, is either unspecified or explicitly left undefined. In addition, the user does not have control over such characteristics as open vs. closed subroutines, and reentrant vs. non-reentrant code.

j. Efficient Object Representations and Machine Dependencies. The major conflicts in this area are that optimizations potentially can change program effect, there is no facility for machine code inserts, and the user has no control over open vs. closed subroutines.

k. Program Environment. The major conflict in this area is that JOVIAL does not have a special facility for including assertions in programs.

l. Translators. There are no major conflicts in this area.

m. Language Definition, Standards and Control. The major conflict in this area is the incomplete and ambiguous specification of the language.

3. Unnecessary Features in JOVIAL.

The following features of JOVIAL are unnecessary in the sense that they are not specifically needed to satisfy any of the Tinman requirements: the option of choosing between parallel and serial organization of tables, the SHIFT, SIZE, and DSIZE functions, unsigned integers, and the fall-through option of the SWITCH statement. We recommend that the

fall-through option of the SWITCH statement be deleted (due to its error-proneness) and the other features named above be retained.

4. Recommendations concerning JOVIAL.

On the basis of the evaluation conducted in this chapter, we conclude that it is inadvisable to modify JOVIAL to satisfy the Tinman. JOVIAL differs substantially from the Tinman with regard to the specific requirements and also the general goals and philosophy. Consequently, it is not reasonable to expect that JOVIAL could be modified to satisfy the Tinman while still adhering to its original design goals.

# CHAPTER 6

# FORTRAN EVALUATION

## Section I. LANGUAGE SUMMARY

### 1. Lexical Properties.

a. FORTRAN is a semi-fixed format, line-oriented language. End-of-line flags the termination of a statement unless a non-zero digit appears in the continuation column, 6, of the succeeding line. Columns 1 through 5 are reserved for an optional numeric label, and the statement itself must appear somewhere within columns 7 to 72.

b. The language is defined using 49 characters, all of which appear in the 64 character ASCII subset. Non-standard symbols (i.e., any an implementation wishes to support) may appear within character data, edit field descriptors and comments.

c. Identifiers may be a maximum of 6 characters in length and must begin with a letter. Spaces are permitted internal to syntactic units as a break character and are ignored, except within keywords and character literals. Keywords are not reserved and their recognition relies on context.

d. Only full-line comments are supported. A "C" or "*" in column 1 signals a comment line.

### 2. Data Types

Figure 5 categorizes the FORTRAN data types.

a. Built-in Data Types.

(1) Introduction. FORTRAN differentiates between two classes of built-in data types according to their storage characteristics. Character variables occupy a fixed number of character storage locations determined by their compile-time specified length. All other scalar data types are allocated either one or two (for complex and double precision) non-character storage locations.

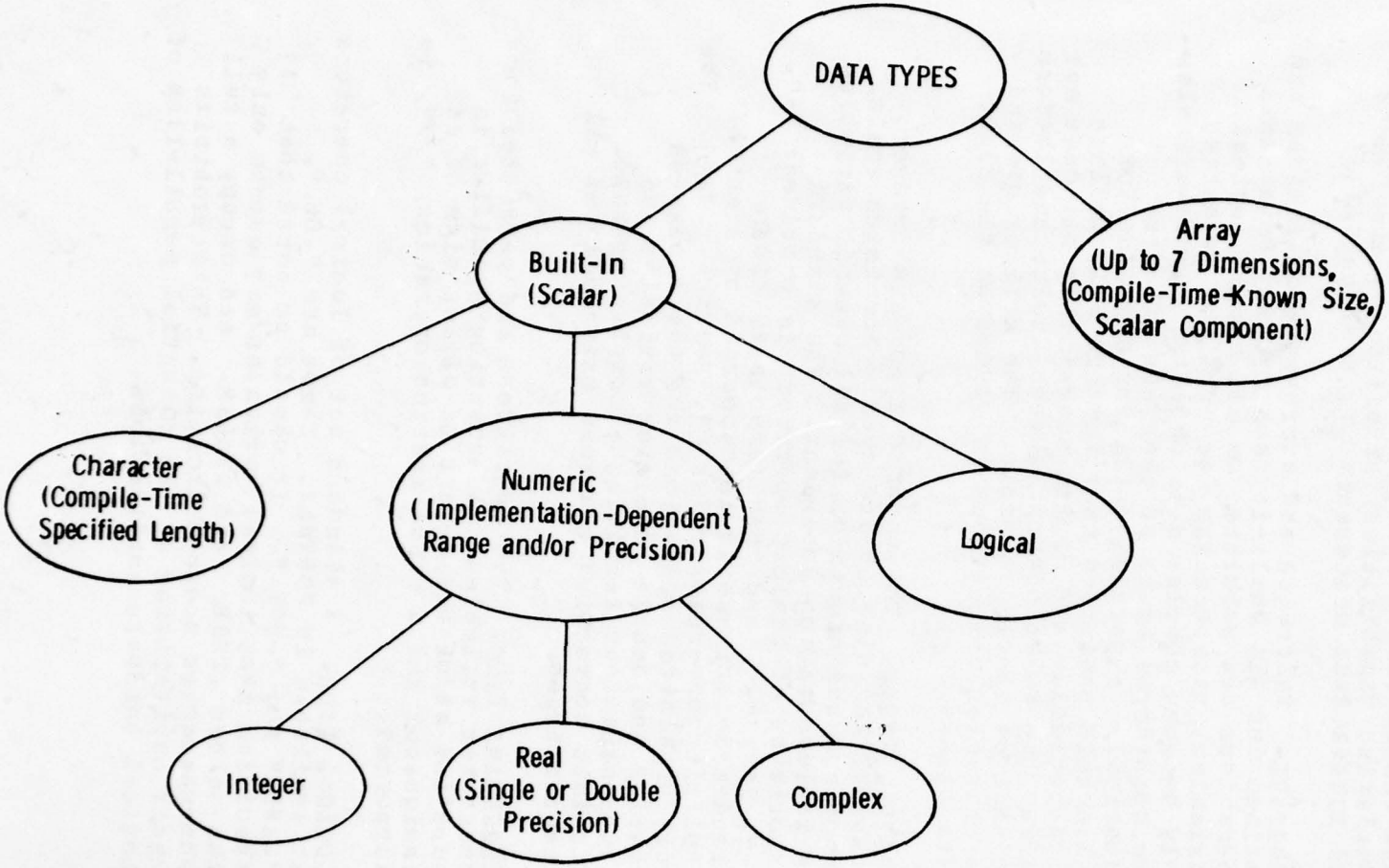



Figure 5. Data Types in FORTRAN

This distinction restricts the associations (i.e., overlaying capabilities) permitted by the COMMON and EQUIVALENCE statements [1, Section 8.3].

(2) Behavior. Reference and assignment operations are defined for all built-in data types. Relational operations are permitted on all except logical variables, with testing for equality permitted only between complex data objects. Real variables are considered equal if they are bit-wise identical. A partial collating sequence for characters specifies that letters will collate alphabetically and digits numerically but does not define the relationship between digits and letters or that of special symbols. The smaller operand in character comparisons is padded on the right with blanks.

(3) Numeric Types. The numeric types are integer, complex, and single- and double-precision reals. The range and precision for all numeric variables is implementation-dependent. The standard assortment of numeric operators is provided: "+", "-", "*", "/", and "**" (exponentiation). Mixed-mode arithmetic is supported by a set of implicit conversion rules [1, Section 6.1.4]. The only prohibited operations are those between complex and double precision variables and exponentiations involving a complex operand. Explicit conversion routines exist between all numeric types.

(4) Character Type. Two operations are permitted for character variables. A substring specifier is supplied which may appear on either side of an assignment and a concatenation operation, "//", is supported.

(5) Logical Type. A standard set of logical operators is available in FORTRAN. These are ".OR.", ".AND." and ".NOT.". It should be noted that [1] specifies that logical variables may assume only the values .TRUE. and .FALSE. and occupy a full non-character storage location. This prohibits their capitalizing on the potential parallelism of logical hardware instructions.

b. Arrays. The only aggregate data structure available in FORTRAN is the array. Arrays must be of compile-time size and dimensionality. The default lower bound on each dimension is 1, but may be overridden by the programmer. The array permits homogeneous composition of any of the built-in data types. Elements of a character array must all be of identical length. Actual array parameters must be equal to or greater than their associated dummy parameters in size and are always passed by reference. Component selection is the only array operation supported.

3. Procedures.

a. Procedure Classes. FORTRAN supports three types of procedures. Statement functions, similar to mathematical functions, are limited to the computation of a single statement and return a value. They are local to the program module (*) in which they appear, may access any variables defined within that program module, may not have side effects, and contain parameters local to their single statement. Function subprograms also return a value but may be of arbitrary length, and can access variables passed to them as parameters, available through COMMON, or defined locally to the subprogram. Subroutines are identical to function subprograms but may not return a value and are invoked by a CALL statement.

b. Parameter Types.

(1) Value parameter. Any built-in type may be passed "by value". The specification of value passing occurs on the calling side of a procedure invocation. For example, the parameters "+R" imply a "by value" or copy-in mechanism, whereas "R" denotes "by reference."

(2) Reference parameter. All data types including arrays may be passed "by reference."

(3) Procedure parameter. Either a function subprogram or subroutine may be passed as a parameter. An EXTERNAL declaration is required for all such

---

(*) A program module is either a main program, function suprogram, or subroutine. The most local scope possible for a variable is a program module. The only exception to this rule is parameters to statement functions.

parameters.

(4) Alternate return specifier. Statement labels may be parameters to subprograms. Optionally, a RETURN statement which is used to transfer control back to the calling program module may specify a positive integer which will cause control to return to the nth label parameter, where n is the value of the integer so specified.

c. Recursion. Recursion is not permitted in FORTRAN.

4. Statements

a. Assignment Statement. The FORTRAN assignment operator is the equals sign, "=". Type compatability is required for all assignments but implicit conversions will be automatically made between numeric types. Array assignment is not available.

b. CONTINUE Statement. The CONTINUE statement causes no action to be taken but can contain a statement label and thereby specify a control point.

c. GOTO Statement. The FORTRAN GOTO permits control transfers to any label local to the containing program module.

d. END Statement. This statement denotes the lexical end of a program module.

e. DO Statement. Iterative execution in FORTRAN is provided by the DO statement. A loop control variable, which is global to the containing program module, is required. Its initial value and termination limit must be specified. An optional step size may be included but will default to 1 if omitted.

f. Computed GOTO Statement. The effect of a case statement is provided in FORTRAN by the computed GOTO. It transfers control to one of a list of labels according to the value of an integer expression.

g. CALL Statement. The CALL statement is used to invoke a subroutine.

h. RETURN Statement. The RETURN statement was discussed in paragraph 3.b(4) of the section.

i. ENTRY Statement. The definition of alternate entry points to subprograms may be done with this statement. ENTRY statements may define dummy parameters in the same manner as subroutine or function specifiers. Appropriate rules exist to prohibit reference to an undefined dummy parameter within subprograms [1, Section 15.7].

j. SAVE Statement. This statement specifies variables within a subprogram which are to retain their values from one invocation to the next. These variables then become identical to "own" variables in ALGOL-60.

k. ASSIGN Statement. The ASSIGN Statement allows the assignment of a statement label to an integer variable.

l. Assigned GOTO Statement. Once a label has been assigned to an integer variable, this statement permits control transfer to that label. Optionally, a list of potential destinations may be included.

m. Conditional Statements.

(1) Arithmetic IF. Depending upon the relationship between a specified arithmetic expression and zero, this statement causes a jump to one of three labelled locations.

(2) Logical IF. This statement allows the value of a logical expression to cause conditional execution of a given statement.

n. EQUIVALENCE Statement. Variables of similar storage type (i.e., character or non-character), including arrays, may be EQUIVALENCEd. Variables so associated share common storage locations.

o. COMMON Statement. The COMMON statement allows inter-routine communication. COMMON blocks simply specify regions of memory upon which each routine is permitted to define any organization. The only restriction that exists is that character and non-character variables may not appear within the same COMMON block or be EQUIVALENCEd (overlayed) by two organizations of a given block

5. Storage Allocation.

FORTRAN is designed so as to allow static storage allocation. The size of all data items, except arrays passed by reference (which do not require memory allocation), is known at compile-time. No recursion or parallel processing is permitted.

6. Files and I/O.

a. File Storage.

(1) Internal files. Each internal file is associated with some area of character storage. This allows I/O to be directed to or from a character datum.

(2) External files. External files reside on an external device identified by an integer unit number. The association of a specific device with a unit number is implementation-dependent.

b. File Types.

(1) Sequential files. Sequential files contain an ordered sequence of records. Each record is stored in some formatted fashion. Records may be of differing formats and are read or written sequentially.

(2) Direct files. Direct (random) access files consist of a set of records of equal size. Associated with each record is a record number which permits retrieval and update to occur in any order.

(3) Stream files. Stream files are unformatted, read sequentially, and treated as a stream of characters.

c. I/O Control.

(1) External files are selected for access by the OPEN statement. Each file can, optionally, have associated with it an implementation-dependent file name. The OPEN statement permits specification of accessing method, record length,

the maximum number of records, treatment of blanks, and file status. The file status may be one of the following: OLD, NEW, SCRATCH, or UNKNOWN. The UNKNOWN value has an implementation-dependent meaning.

(2) Data transfer is accomplished by the READ, WRITE, and PRINT statements. These statements may include a unit number and format reference. The absence of a unit number has an implementation-dependent meaning. If the format specifier is omitted, list-directed formatting is implied. List-directed formatting is available only for stream files. (See [1, Section 13.6] for the details of this formatting.)

(3) Three statements are provided for file positioning: BACKSPACE, REWIND, and ENDFILE. Only the ENDFILE statement is available for use with direct files.

7. Process Scheduling.

FORTRAN contains no facilities for process scheduling.

8. Exception Handling.

The exception handling facilities available in FORTRAN are limited to error conditions identified during input and output. FORTRAN I/O statements permit the optional inclusion of error labels. The situations in which this error path will be taken are implementation-dependent. No facility is defined in [1] for providing information to the program about the type of error encountered. The action taken in the event of any non-I/O error is entirely implementation-dependent.

9. Compile-Time Facilities.

[1] does not define any compile-time facilities. The language contains no macros, compile-time variables, or conditional-compilation capabilities. A separate compilation facility is implicit in the EXTERNAL statement but is left undefined.

## Section II. DATA AND TYPES

1. Typed Language (A1).

a. Degree of compliance: P

b. The types of all FORTRAN data objects are determinable at compile-time and are unalterable at run-time. However, explicit type declarations are not required. The implicit specification is that symbols beginning with the letters I, J, K, L, M, N represent integer variables and all others denote floating-point variables. The programmer may both redefine this implicit assumption and override it through explicit declarations.

c. The notion of "type" in FORTRAN is somewhat limited. For example, the type of an array is taken to be the type of the underlying component (integer, real, etc.) and does not include the dimension sizes or even the number of dimensions. Thus FORTRAN does not prohibit the programmer from passing an array argument to a routine that expects a differently structured array parameter. As stated in [1, Section 15.9.3.3], "the number and size of dimensions in an actual argument array declarator may be different from the number and size of the dimensions in an associated dummy argument array declarator."

d. In general, FORTRAN does not require the translator to validate the type compatibility of associated (*) data items. "A processor assumes that the type specified for the name agrees with the type of the data" [1, Section 20.4]. This statement demonstrates the weakness of FORTRAN's type-checking. The responsibility for adherence to type rules rests with the programmer.

e. The implicit assumptions made by FORTRAN may be easily removed. The mechanisms for explicit type declarations already exist. The only change required involves reporting undeclared variables. This simply implies modification of the symbol table maintenance routines.

---

(*) Two data items are said to be <u>associated</u> if any component of one shares a common storage location with some component of the other. The mechanisms which cause association in FORTRAN are parameter passing, COMMON and EQUIVALENCE statements.

f. The strengthening of the concept of array type in FORTRAN may be accomplished fairly easily. It will, however, require some runtime checking to be done. This is necessary when array dimensions are specified dynamically (i.e., in terms of some parameter) in a subprogram. This will eliminate some common programming tricks of the language, but should not severely limit its programming power, especially if array operations are permitted (see our discussion in connection with B1).

g. The validation of type compatibility for all associated data items requires extensive modification of the language.

(1) All formal and actual parameters must be checked for compatibility. To support full compile-time checking would necessitate further specification of procedure parameters. The EXTERNAL and INTRINSIC declarators need to be expanded to include definition of the type and number of parameters for the associated procedures.

(2) The COMMON and EQUIVALENCE statements would be the source of the greatest problems in regard to strong type-checking in FORTRAN. One has basically three options as to how to overcome these problems. These statements can be totally eliminated from the language; full type-checking can be performed on them; or some compromise can be made regarding their type security. If removed from the language these statements could be replaced by secure access and overlaying facilities. Strong type-checking would ensure the security of these statements but limit their use for overlaying purposes. One possible compromise would be to limit the access scope of differently-typed associated items and enforce the restrictions on the accessing of these variables before assignment of the proper type.

2. Data Types (A2).

a. Degree of compliance: P

b. FORTRAN provides data types for integer, floating point, Boolean and character [1, Chapter 4] and provides

arrays of these types [1, Chapter 5]. There exists no general fixed-point type besides integer, and there exists no record definition facility, a serious drawback.

c. The feature of FORTRAN which most closely approximates a record definition facility is the named COMMON statement [1, Section 8.3]. Several modifications could be made to transform this statement into a rudimentary record declarator; however, these would significantly change the intent of this feature.

d. The introduction of a record definition facility in FORTRAN would require extensive additions to the language. While these changes would not necessarily prompt modification of existing features, they would certainly complicate their implementation.

e. A record consists of the composition of a finite number of components, each of some predefined type and size. The record itself and each component has an associated name by which it can be referenced. For consistency with E1, the declaration of a record should not require any storage allocation, but should simply serve as a template for variables which are later defined to be of that type (i.e., the declaration of a record defines a new type).

f. The definition of a record facility will interact with the existing COMMON and EQUIVALENCE statements. If components within a record can be EQUIVALENCEd, type-checking will be complicated but alternate structures could be so supported (see A7 for further discussion). At present, the use of character variables in common blocks is restricted. "If a character variable or character array is in a common block, all of the entities in that common block must be of type character" [1, Section 8.3.1]. This restriction stems from a need to prevent implementation-dependent common block associations between numeric and character variables. Strong type-checking on COMMON variables (or their elimination) would circumvent this problem. Otherwise, an implementation-independent character representation and some functional relationship between character and non-character storage sizes could be defined.

g. The introduction of a general fixed point facility is discussed in A4.

3. Precision (A3).

a. Degree of compliance: F

b. FORTRAN offers "REAL" and "DOUBLE PRECISION" as the only floating-point precision specifications. Both have a hardware-dependent definition [1, Section 4.5]. The precision may be defined for a program unit by using the IMPLICIT definition to specify all variables of floating-point type to have single or double precision. This implicit specification may be overridden in explicit type specifications.

c. To bring FORTRAN into partial compliance with this requirement would be relatively easy. It could be accomplished simply by requiring a statement of the intended meaning of single and double precision in each program. This would promote portability and necessitate only minor modification of the language (the inclusion of one new statement) and the translator (at least, informing the user if the machine cannot support that precision).

d. The inclusion of an ability to further delineate precision specification for all variables would be slightly more difficult but still quite feasible. The rules for evaluation of mixed precision expressions already exist [1, Section 6.1.4]. Either existing type declarators would have to be extended or a new precision specifier added to permit further precision definition. The translator could simply use this precision definition to decide whether a variable should be represented in conventional single or double precision. This would involve an extremely minor change.

4. Fixed Point Numbers (A4).

a. Degree of compliance: F

b. FORTRAN does not offer any fixed-point data type except integers. Integers are treated as exact quantities with a step size equal to one.

c. The introduction of a general fixed-point data type would require a large effort. The language would need to be expanded to include a fixed-point declarator. If fully integrated into the present language, fixed-point variables, just as those of any other numeric type, would be permitted

to appear anywhere in arithmetic expressions. This would require more general mixed-type expression evaluation rules, additional intrinsic conversion routines, and compiler and run-time scaling procedures. Run-time scaling would be required when assigning floating-point values to fixed-point variables. In general, mixed-mode arithmetic would be greatly complicated, and possibly large efficiency losses, which are hidden from the programmer, could result. Another option is the elimination of mixed-mode arithmetic (see our comments concerning B8).

5. Character Data (A5).

a. Degree of compliance: F

b. FORTRAN makes the following restrictions on the collating sequence [1, Section 3.1.5]. "A" is less than "Z" and letters collate in alphabetic order; "Ø" is less than "9" and digits collate in numerical order; blank is less than "Ø" and less than "A". There are no other constraints on the collating sequence. This means that numeric characters may collate before (USASCII) or after (EBCDIC) alphabetic characters, and the order and position of special characters relative to alpha or numeric characters is undefined. There is a recommendation offered: "A processor, if possible, should use the ASCII collating sequence for the complete FORTRAN character set" [1, Section 20.3]. FORTRAN offers neither standard translate routines nor alternate character sets.

c. The new draft proposal for FORTRAN improves FORTRAN's handling of character data, but the language still falls far short of meeting this requirement. To bring FORTRAN into total compliance would be a difficult task. FORTRAN does not offer any ability to define enumeration types. Even with the addition of such a facility (see B3), many problems still remain. For example, character variables and arrays are permitted to be used as format specifiers [1, Section 13.1.2]. The introduction of user-defined character sets would then greatly complicate input-output and cause the utilization of character variables as format specifiers to incur additional runtime costs.

d. Partial compliance can be gained, however, through a simple addition to the language. A statement should be added which permits the definition of any underlying

ordering assumptions that are being made in addition to those of the defining document [1]. Requiring use of this statement would improve both portability and readability.

6. Arrays (A6).

a. Degree of compliance: P

b. FORTRAN requires user specification of the number of dimensions [1, Section 5.1.5] and the range of subscript values of each dimension [1,Section 5.1.1.1]. The type of each array component may be specified implicitly [1, Section 5.2.1]. The number of dimensions and the array type are determinable at compile time. Subscript bounds must be known at compile time for arrays defined in the main program; the bounds for dummy arrays (i.e., formal parameters) may be specified at run time. FORTRAN arrays are allocated at compile time (or before execution of the main program). The default lower bound of an array is 1 [1, Section 5.1.1.2]. The range of subscript values for any given dimension is a contiguous subsequence of integers [1, Section 5.2].

c. Permitting the allocation of arrays at scope entry would entail major language modifications. The basic flavor of FORTRAN is one of static memory allocation. No features presently exist that require dynamic memory management. The addition of any such features would reduce runtime efficiency and greatly complicate the use of some existing facilities. The type-checking required by A1 would have to be done at run time. The correspondence of common block size would also have to be checked during execution. EQUIVALENCE statements could no longer be validated during compilation. These problems with COMMON and EQUIVALENCE statement stem from their storage orientation.

7. Records (A7).

a. Degree of compliance: F

b. FORTRAN has no facility for defining records (i.e., heterogeneous data structures with named fields); thus there is no provision for records with alternative structure. The way in which the facility called for in this requirement is achieved in FORTRAN is in fact the EQUIVALENCE statement prohibited by A7. The problem with EQUIVALENCE is that it

facilitates the defeat of type checking, as mentioned earlier in connection with A1. In addition, the feature has various restrictions [1, Sections 8.7.5 and 8.3.6] because of its storage orientation and the interactions with other features such as COMMON.

c. The introduction of an extremely primitive record facility was discussed in A2. The use of this facility in conjunction with a strongly-typed EQUIVALENCE statement permits safe use of alternate structures within records. The necessity of type compatibility could be removed if compile-time type-checking rules were relaxed. Performing the type-checking at run time would require the maintenance of a tag-field. The easiest way in which alternate structures could be supported would require all such structures to have the same size. This size restriction simply necessitates the allocation for the largest possible record. This would permit records to be stored in common without any further changes. Another alternative is dynamic allocation and common validation, which was shown to be a fundamental change in the language in A6.

## Section III. OPERATIONS

1. Assignment and Reference (B1).

a. Degree of compliance: P

b. FORTRAN defines assignment and reference operations [1, Sections 10 and 2.12] for all scalar data types. However, assignment is not provided for arrays. Moreover, the absence of encapsulated type definitions implies that no user-defined assignment is permitted. It should also be pointed out that the label assignment allowed in FORTRAN (the ASSIGN statement [1, Section 10.31] is generally considered poor programming practice, since it serves to obscure the behavior of the program.

c. Array assignment poses no real difficulty for FORTRAN. Arrays are of compile-time-known size and dimensionality, except for adjustable arrays [1, Section 5.5.1] within subprograms which will be required to be of known dimensionality if the suggestions of A1 are implemented. This then leaves only the addition of runtime tests to check size compatibility when dealing with adjustable arrays and of code generation routines for array assignment.

d. Record assignment involves problems similar to those of array assignment. The required additional effort for its implementation depends upon how records are defined (see our discussions regarding A2 and A7). The ability to have run-time-determined alternate structures of different sizes would complicate assignment and reference operations.

e. The introduction of user-defined assignment for encapsulated type definitions is discussed in E5.

2. Equivalence (B2).

a. Degree of compliance: P

b. The FORTRAN comparison operator will compare any two scalar operands of arithmetic type or any two scalar operands of character type. If arithmetic operands are of different types then integer is converted to real or double precision, or real is converted to double precision. Two floating-point values are equal only if they are bit-by-bit

the same. FORTRAN does not provide a record data type and arrays may only be compared element by element.

c. What is required to bring FORTRAN into full compliance with this requirement is to extend comparisons to include array and record operands and to modify the equality rules for floating point numbers. In the case of arrays and floating-point numbers, meeting these requirements would entail only a small effort. Record comparison is somewhat more complex, and if alternate structures are to be supported, these comparisons would involve greater run-time and compile-time overhead. The inclusion of precision specifications as proposed in A3 would be the only design modification required to support limited-precision identity for floating-point numbers.

3. Relationals (B3).

a. Degree of compliance: P

b. All six relational operators are defined for numeric data and for character data [1, Section 6.3]. FORTRAN provides no facilities for definition by enumeration.

c. The inclusion of facilities for definition by enumeration requires extension of FORTRAN, but poses no great difficulties if strong type-checking is maintained. Elements of enumerated types can be represented as integer quantities denoting their position in the enumeration. This permits all six relational operators to be applied in exactly the same manner as they would be for integers.

4. Arithmetic Operations (B4).

a. Degree of compliance: PT

b. Aside from lacking integer division with a real result, FORTRAN provides the required operations [1, Section 6.1]. The remainder of integer division is calculated with the MOD intrinsic function [1, Section 15.10]. Floating point operations are as precise as the operand of greater precision [1, Section 6.1.4].

c. Integer division with a real result could very easily be explicitly defined in the language, but it is already indirectly available. By first converting the operands to real numbers, one is able to compute the required division.

d. The present division operator could be redefined to return a real result with integer operands; however, this could lead to great confusion. Historically, FORTRAN has truncated the result of such an operation, and many programmers are accustomed to using this feature.

5. Truncation and Rounding (B5).

a. Degree of compliance: PT

b. FORTRAN satisfies this requirement, except for the lack of information on the behavior of the program in the presence of arithmetic overflow. It is possible for such an overflow to result in high-order truncation with no message to the user that this has occurred.

c. Overflow reporting is dealt with as part of the general discussion of exception handling in G7.

6. Boolean Operations (B6).

a. Degree of compliance: P

b. The FORTRAN built-in Boolean operations include "and", "or", and "not", but "nor" is not provided [1, Section 6.4]. FORTRAN allows but does not require short-circuit mode in the evaluation of "and" and "or" [1, Section 6.6.1].

c. The addition of a "nor" operator to the language is trivial. It simply requires an addition to the table of relational operators in [1, Section 6.4.1] and a few new lines of code to the compiler. The strengthening of wording in [1, Section 6.6.1] to require short-circuit mode evaluation will not only bring FORTRAN into full compliance but will also simplify the language. As it stands, not only can the programmer not depend on short-circuit evaluation, but he must also accept the fact that variables assigned to as a side effect of a possibly short-circuited expression become undefined after evaluation of that expression [1, Section 17.3].

7. Scalar Operations (B7).

a. Degree of compliance: P

b. FORTRAN does not permit any operations on data aggregates except I/O operations [1, Sections 6.1.2.1 and 10.1]. All operations required in B7 must be explicitly programmed element by element. FORTRAN does not provide records.

c. Record definition is discussed in A2 and A7. Assignment of data aggregates is reviewed in B1.

8. Type Conversion (B8).

a. Degree of compliance: P

b. FORTRAN converts implicitly among arithmetic types in expressions and for assignments [1, Section 6.1.4]. FORTRAN does not convert an argument to the parameter type; when arguments and parameters differ in type the program is in error and this error need not be detected [1, Section 15.5.2.2 and 15.6.2.3]. FORTRAN does provide all the required explicit conversion operations using intrinsic functions among arithmetics [1, Section 15.10] and I/O operations between object and character representations or arithmetic data.

c. The type-checking proposed in A1 would guarantee the reporting of mismatched actual and formal parameters. FORTRAN could be modified fairly straightforwardly so as to remove all implicit conversions. The routines for all valid explicit conversions exist. However, this would be a fairly radical departure from the flavor of the language.

9. Changes in Numeric Representation (B9).

a. Degree of compliance: P

b. FORTRAN does not provide any facilities for restricting the numeric ranges offered by a hardware implementation; therefore, conversion will never be required. There is no provision for detecting integer overflow or floating point overflow or underflow.

c. Range specifications can be added to FORTRAN. No features of the language directly prohibit this, but if the compiler is meant to use these specifications to optimize memory utilization, several features will be impacted. If the storage size for variables is to be compiler-determined,

then statements which are storage oriented (i.e., EQUIVALENCE and COMMON) must be handled delicately. Integer and fixed-point truncation tests could be optionally compiled into the object code and when truncation occurs processed like any other exception condition (see G7).

10. I/O Operations (B10).

a. Degree of compliance: P

b. FORTRAN provides operations allowing programs to interact with files, channels and devices including terminals [1, Chapter 12]. Data (and control information in the form of data) may be sent and received. FORTRAN also provides some explicit I/O control functions [1, Section 12.10]. The language allows user control over some exception conditions; however, the definition of an exception condition is implementation dependent, and whether or not the user retains control of all exception conditions is an implementation decision. The form of file names and the possible characteristics of files are implementation dependent.

c. To fully comply with requirement B10, greater standardization is required in FORTRAN I/O. A full definition of the OS interface is needed. This would necessitate further work on the language definition and an additional large effort in compiler writing, especially if the possibility of running with no operating system is allowed for.

d. A small modification specifying that all errors would transfer control to the error label in an I/O statement and requiring that such a label exist would bring FORTRAN into greater compliance with this requirement. Additionally, some mechanism for returning the nature of some otherwise undefined error should exist. Both these features could be incorporated into the language with small modifications to the defining document and a reasonable amount of added effort in compiler implementation.

11. Power Set Operations (B11).

a. Degree of compliance: P

b. FORTRAN provides no facility for defining enumeration types; however, the FORTRAN logical array can be used to provide set operations.

c. The primary difficulty with using logical arrays to simulate power sets is that an element of such an array occupies a full noncharacter storage cell [1, Section 4.7]. This precludes the use of logical machine instructions for parallel processing of these operations. Either of two options can be chosen to overcome this problem. A logical array can be defined to be optionally packed or a set declarator can be added to language. The use of a packed logical array, while requiring less language modification, would probably lead to implementation-dependencies. This would be due to the dictating of array size by machine word size (e.g., a 36 element set would occupy 2 words on an 18-bit machine and only 1 on a 36-bit machine).

## Section IV. EXPRESSIONS AND PARAMETERS

1. Side Effects (C1).

a. Degree of compliance: F

b. FORTRAN explicitly prohibits certain side effects [1, Section 6.6]:

> "The execution of a function reference in a statement may not alter the value of any other entity within the statement in which the function reference appears. The execution of a function reference in a statement may not alter the value of any entity in common ... that affects the value of any other function reference in that statement."

Other side effects, such as overflow, are ignored. Moreover, there is no guarantee that occurrences of the prohibited side effects will be detected in programs. There is no guarantee that side effects will be evaluated in left-to-right order, especially since the order of evaluation in expressions is left undefined.

c. Satisfaction of this requirement within FORTRAN would affect only the optimization section of the compiler. However, in this area it would have tremendous impact. Because of the difficulty of detecting all side effects, many optimizations commonly applied to FORTRAN programs would be eliminated. This could have significant impact on run-time efficiency.

d. If this requirement is truly the desired goal then the restrictions can be easily included in the language definition. Compiler implementation need be no more complex, since potential optimizations can be ignored if their application might violate C1.

2. Operand Structure (C2).

a. Degree of compliance: PT

b. FORTRAN has few levels of operator hierarchy, and the ones present are widely recognized. However, FORTRAN does not require explicit parentheses "when the execution order

is of significance to the result within the same precedence level". For example, X/Y/Z, X/Y*Z, and X**Y**Z are legitimate expressions, with interpretations (X/Y)/Z, (X/Y)*Z, and X**(Y**Z), respectively. The FORTRAN user is not able to define new operator precedence levels nor to change the precedence of existing operators.

c. The only required modification here is minor. The rules for evaluating unparenthesized expressions containing operators of equal precedence should be dropped and the language definition should state, "when the order of evaluation is not determinable from operator precedence and can be significant to the result, the expression must be more fully parenthesized". Checking for this condition would constitute a fairly trivial modification to the compiler.

3. Expressions Permitted (C3).

a. Degree of compliance: T

b. FORTRAN completely satisfies this requirement, correcting the deficiencies (cited in the Tinman) which were characteristic of previous versions of the language.

4. Constant Expressions (C4).

a. Degree of compliance: P

b. FORTRAN allows constant expressions in programs anywhere that constants are allowed. The FORTRAN standard does not require constant expressions to be evaluated at compile time.

c. The inclusion of a rule that constant expressions must be evaluated at compile-time entails little additional effort. The impact on compiler design is relatively small. Even in cases of cross-compilation, as compared to the overall compiler development, this necessitates a reasonably small amount of work. The ease with which this requirement can be met stems from its independence from other language features.

5. Consistent Parameter Rules (C5).

a. Degree of compliance: P

b. FORTRAN does not have exception handling or a parallel processing capability. The intrinsic functions of FORTRAN obey the same parameter-passing rules as user-defined functions. There are no operations applicable only to parameters.

6. Type Agreement in Parameters (C6).

a. Degree of compliance: P

b. FORTRAN requires that the type of each argument agree with the type of the corresponding parameter. The bounds of an array may be passed explicitly as arguments to a procedure. Whether a type transfer hidden in a procedure call is or is not detected, and the result of any operations involving such type transfer, are implementation dependent. In addition, FORTRAN's restrictions on array passing are fairly weak [1, Section 5.5]: "If the actual argument is an array name, the size of the dummy array must not exceed the size of the actual argument array."

c. The language and compiler modifications required for strengthening parameter type checking are discussed above in connection with requirement A1.

7. Formal Parameter Kinds (C7).

a. Degree of compliance: P

b. FORTRAN partially satisfies this requirement. Scalars may be passed either by value or by reference [1, Section 15.9.3]; however, the kind of passage is specified on the calling side and not with the formal parameter. For example, A is passed by reference in CALL SUB(A) and by value in CALL SUB ((A)) or (if A is numeric) in CALL SUB(+A) [1, Sections 20.6 and 15.9.3]. Arrays may only be passed by reference. FORTRAN provides procedure parameters and also "alternate return specifiers" which may be used to control action when user-detected exceptions occur.

c. The ability to pass composite structures (i.e., records and arrays) by value would require a good amount of

run-time support. Since arrays may not be of compile-time determined size (in the case of adjustable arrays), some dynamic memory management would be required. This leads to the problems discussed above in connection with requirement A6.

8. Formal Parameter Specifications (C8).

a. Degree of compliance: F

b. FORTRAN fails to satisfy this requirement. Any procedure parameters which are not explicitly declared are specified by default and do not provide the required facility.

c. It would be possible to support generic procedures in FORTRAN; however, several major problems would arise from their inclusion. Some mechanism must be established for their definition. This mechanism, while potentially syntactically quite similar to normal subprograms, would be semantically quite different. Semantically, it is most like a macro definition (but with only limited replacement and with only implicit as opposed to explicit information transfer).

d. This feature could be included in the language with a moderate amount of language redesign and a large degree of compiler support.

9. Variable Number of Parameters (C9).

a. Degree of compliance: F

b. As stated in [1, Section 15.9], "The number of actual arguments must be the same as the number of dummy arguments in the procedure referenced." FORTRAN has no provision for variable numbers of parameters, except for the built-in ("intrinsic") functions MAX and MIN.

c. To provide the suggested facility would require a moderate amount of work. The method by which arrays are passed in FORTRAN somewhat simplifies the task. Since "the size of an adjustable array must not exceed the size of the actual argument array," the mechanism for accepting a variable number of elements without setting any maximum already exists. However, the consequence of using this

array passing mechanism is that all arguments so specified appear as by-reference parameters at procedure entry. On the calling side, however, since arrays must be stored contiguously in memory and the parameter variables as declared may occupy non-contiguous locations, a block of storage must be allocated and the values copied into it. This implies, despite the apparent by-reference passing on the procedure side, that by value parameter passing needs to be used. The aforementioned allocation does not necessitate dynamic memory management since the actual number of parameters is known at compile time.

## Section V. VARIABLES, LITERALS AND CONSTANTS

1. Constant Value Identifiers (D1).

a. Degree of compliance: PT

b. FORTRAN satisfies this requirement to a high degree [1, Section 8.6]; the only exception is that there is no provision for constant arrays.

c. The inclusion of an array literal as discussed in D2 and the ability to use it within the PARAMETER statement [1, Section 8.6] will bring FORTRAN into total compliance with this requirement. This constitutes a relatively minor modification and incurs no run-time costs.

2. Numeric Literals (D2).

a. Degree of compliance: P

b. FORTRAN provides a syntax and consistent interpretation for constants of built-in data types [1, Sections 4.3 through 4.8]. The relationship between values of numeric literals and input or output data is not considered in [1].

c. If array assignments as reviewed in B1 are permitted, only a small expansion of the language to include array literals is needed. The requirement for equivalence between numeric literals and input data must be added to [1]. This introduces the difficulties mentioned in the Tinman when cross-compilation is involved but the additional work relative to the overall development of a cross-compiler is small.

3. Initial Values of Variables (D3).

a. Degree of compliance: P

b. FORTRAN permits the user (via the DATA statement [1, Chapter 9]) to specify the initial values of individual variables. The initialization specification is separate from the declaration and, for COMMON variables, may appear in a different program unit [1, Chapter 16]. All FORTRAN variables are preallocated; thus, variables are initialized at the time of their apparent allocation. There are no

default initial values [1, Section 2.11]. FORTRAN has no provision for testing for variable initialization.

c. The ability to initialize variables as part of their declaration requires a syntactic modification of FORTRAN; however, it should only shift where the work is done in the compilation not increase it. This necessitates moderate language redesign and only small compiler changes.

d. Detection of potentially undefined references at compile time and run-time testing for initialization would increase the complexity of the compiler but would not change the nature of the language. [1] states, "an entity must be defined at the time a reference to it is executed." The additions necessary to enforce this restriction would require greater work during compiler development and be somewhat complicated by the COMMON and EQUIVALENCE statements. This would be particularly true if these statements are permitted to associate variables of differing types.

4. Numeric Range and Step Size (D4).

a. Degree of compliance: F

b. FORTRAN does not require its users to specify the range of numeric variables nor does it provide any mechanism for such specifications. The only range limitations in FORTRAN are those inherent in the hardware; the results of any operations exceeding hardware limitations are undefined [1, Section 1.2.2].

c. The scope of modifications needed to support a fixed-point number facility is discussed above in connection with requirement A4. The introduction of a mandatory range specification for all numeric variables can be accomplished without adversely impacting other language features. However, its use for optimization of storage allocation would cause intricate problems in regard to the COMMON and EQUIVALENCE statements. These problems could be minimized by a redesign of these statements in such a manner as to make them less storage oriented. This would entail a large amount of effort in both language redesign and compiler development.

5. Variable Types (D5).

a. Degree of compliance: P

b. FORTRAN provides neither type-definition facilities, record data structures, nor enumeration types. Arrays and variables may be of any built-in data type; the number of array dimensions is limited to 7 [1, Section 5.1.1].

c. Full compliance with D5 requires extensive language modifications. Some of the issues involved have already been discussed (records in connection with A2 and A7, enumeration types in connection with B3). The ability to define arrays of any component type would complicate the subscripting rules as described in [1, Section 5.4]. Removing the restrictions on the dimensionality of arrays would be a relatively minor modification.

6. Pointer Variables (D6).

a. Degree of compliance: F

b. FORTRAN has no true pointer facility. The EQUIVALENCE statement provides some of the attributes of such a facility (e.g., shared substructure, acting as an additional reference to a datum) but due to its storage orientation is quite different in nature. Many of the issues regarding pointer variables concern features not present in FORTRAN (e.g., composite types). The ease with which pointers can be introduced depends on how these features are designed. A clean implementation should facilitate the introduction of pointers; however, this is a major modification to both language and implementation.

## Section VI. DEFINITION FACILITIES

1. User Definitions Possible (E1).

a. Degree of complaince: F

b. FORTRAN is quite weak in the area of definition facilities. The language permits functions and subroutines to be defined by the user, but there is no provision for defining new infix operators. The data structuring mechanisms are limited; there are no capabilities for user-defined data types, and the only data structuring facilities available are the array and the EQUIVALENCE statement.

c. Many features needed for data definition have already been discussed (records in connection with A2 and A7, enumeration types in connection with B3, and composite component arrays in connection with D5). As can be seen, these modifications alone constitute extensive changes. To reach full compliance with E1 would require a total language redesign. A data definition facility interacts with almost all other language features (e.g., parameter passing, reference and assignment operations) and requires constant attention throughout language definition.

d. The effect of such extensive modifications on implementation would be twofold. The compiler necessary to support these new features would be several times as complex as present compilers, increasing both the cost and time of development. The size of the resulting compiler would severely limit the number of machines on which it could be supported.

e. The specific aspects of data definition introduced in Tinman requirement E1 will be dealt with in the other paragraphs of this section. However, let us mention here that their dependence on the aforementioned language design is great.

2. Consistent Use of Types (E2).

a. Degree of compliance: F

b. Since FORTRAN contains no type definition facilities as discussed in E1, it fails to comply with this

requirement. Therefore, in terms of modification this reduces to an added condition on the redesign necessary to satisfy E1. As such the additional effort relative to the major modifications already needed is small. Within the scope of the present language, facilities for performing operations on arrays are required. These have been discussed in B1 and B2.

3. No Default Declarations (E3).

a. Degree of compliance: PT

b. The FORTRAN default (symbols beginning with the letters I, J, K, L, M, or N stand for integer variables or functions, and all other symbols stand for single precision real variables or functions) is well documented in the language standard and widely known. It is, however, a default. Symbols whose type differs from these defaults must be explicitly declared. All other program components must be explicitly defined.

c. The modifications necssary for removal of FORTRAN's implicit type defaults are described in A1.

4. Can Extend Existing Operators (E4).

a. Degree of compliance: F

b. FORTRAN offers no such facilities.

c. This requirement constitutes an added goal of the language redesign required by E1. Operator extension is, however, somewhat more limited in scope than the general changes described above. If mixed mode expression evaluation is eliminated from the language (as required by B8) and strong type checking is performed at compile time (as required by A1), the scope of additional modifications is fairly small and no large problems should be encountered during implementation.

5. Type Definitions (E5).

a. Degree of compliance: F

b. FORTRAN offers no facilities for defining new types and thereby totally fails to meet this requirement. The

scope of modifications needed to support such facilities is prohibitive (see our discussion above concerning E1) and would be further complicated if compliance with E5 is desired.

6. Data Defining Mechanisms (E6).

a. Degree of compliance: F

b. FORTRAN does not offer any mechanism for defining new types by enumeration, by discriminated union, or as power sets of enumeration types. The Cartesian product of existing types is provided in the array definition facility (to a maximum of seven dimensions). The effect of power sets can be partially realized in FORTRAN via logical arrays.

c. The introduction of general type definition facilities has been considered in connection with E1. Several of the mechanisms suggested in E6 have been discussed in regard to FORTRAN in other sections of this report (records in connection with A2 and A7, enumeration types in connection with B3 and use of logical arrays to simulate power sets in connection with B11). Discriminated union could be supported but would greatly complicate type checking and necessitate some run-time type validation.

7. No Free Union or Subset Types (E7).

a. Degree of compliance: P

b. The EQUIVALENCE statement in FORTRAN has the effect of a free union feature. FORTRAN has no subsetting provisions.

c. The problems associated with the EQUIVALENCE statement have been discussed in connection with A7.

8. Type Initialization (E8).

a. Degree of Compliance: F

b. Lacking type definition, FORTRAN provides no facilities for meeting this requirement.

c. This requirement is actually an extension of the facilities required by E5. The problems associated with its realization are similar to those already discussed above in connection with requirement E5.

## Section VII. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed (F1).

a. Degree of compliance: P

b. FORTRAN offers most of the namescope control required by F1. Variables may be defined local to some subprogram and either be saved from one activation to another or independently allocated for each call. However, FORTRAN is not a block-structured language; thus, a name is accessible throughout the entire subprogram in which it appears. Statement-function parameter names are an exception to this rule; they are accessible only in the single statement-function statement [1, Section 2.9].

c. FORTRAN provides the COMMON statement to allow the definition of inter-routine access rights. Named common statements permit the programmer to limit access to a specific set of routines. This mechanism has at least one great weakness: the common name identifies some region of memory but not the organization of that region. The associated type-checking problems have been discussed in connection with A1.

2. Limiting Access Scope (F2).

a. Degree of compliance: F

b. In FORTRAN, variables are either local or potentially global. No mechanism is available for limiting accessing privileges. Once a variable is defined in COMMON, any and all routines can obtain access to it. The optional inclusion of COMMON declarations allows limiting of access on the call side. The mechanism by which COMMON statements are declared also permits the local renaming of variables. However, the thrust of this requirement is in the direction of user-defined types, and due to FORTRAN's lack of type definition facilities it fails to comply. Inclusion of the ability to redefine substructure names and specify limited global accessing rights with regard to the COMMON statement would entail a reasonable effort. The implementation of such facilities would be non-trivial.

3. Compile Time Scope Determination (F3).

a. Degree of compliance: P

b. Identifiers need not be declared if the user accepts the FORTRAN defaults. Multiple use of variable names is not allowed in the same scope. Variable names may be the same as named-COMMON names [1, Section 20.8]. FORTRAN keywords are not reserved and may, in some contexts, be taken for variable names. FORTRAN is not a block-structured language. Subprograms are always an access scope and the most local definition applies to identifiers. The scope of identifiers is wholly determined at compile time.

c. The reservation of keywords and forced uniqueness of common and variables names constitute a minor change in the language definition. In fact, if keywords become reserved, compiler implementation would be simplified.

d. The changes necessary to make FORTRAN block structured ("lexically embedded scope rules") would be fairly large and involve extensive additions to the compiler. Impact on other language features would be primarily restricted to the common statement. Some consistent and convenient rules would have to be developed to define the interactions between this statement and the block structure.

4. Libraries Available (F4).

a. Degree of compliance: T

b. FORTRAN offers a large set of mathematical functions, and specialized application-oriented subroutine libraries may be made available for FORTRAN. These libraries are outside the scope of the FORTRAN standard definition, but there is nothing in the language which prevents their development.

5. Library Contents (F5).

a. Degree of compliance: F

b. The FORTRAN definition [1] does not directly specify a library facility. The EXTERNAL and INTRINSIC statements do, however, imply that some sort of separate compilation facility must exist.

c. The inclusion within [1] of a definition of a library feature would entail only a small amount of work. The necessary additions to provide compiler support are fairly minor due to the simplicity of FORTRAN's parameter passing and subroutine interfaces. The additions would have no significant impact on existing language facilities.

6. Libraries and Compools Indistinguishable (F6).

a. Degree of compliance: F

b. FORTRAN has no compool facility, nor are compile-time libraries provided.

c. The addition of compools is dependent upon the manner in which data definition facilities are made part of the language (see our discussion above concerning E1). The addition of libraries and compools should be fairly simple in the context of present language facilities (see our discussion above concerning F5), but interaction with other needed additions for Tinman conformance could cause complex problems.

7. Standard Library Definitions (F7).

a. Degree of compliance: P

b. The FORTRAN user has two general I/O statement forms: READ and WRITE. Additional I/O statements such as OPEN, CLOSE, BACKSPACE, REWIND, and ENDFILE are valid only for some kinds of files and devices. File processing may be either stream (character-oriented) or record (record- or block-oriented). Any device which has requirements not satisfiable with these operations requires special-purpose subroutines written for its service.

c. These issues are addressed in our discussion of requirement B10.

## Section VIII. CONTROL STRUCTURES

1. Kinds of Control Structures (G1).

a. Degree of compliance: P

b. FORTRAN provides mechanisms for sequential, conditional, and iterative control. Recursive control is not allowed [1, Section 15.5.3]. No control structures exist in FORTRAN for parallel processing (pseudo or otherwise), asynchronous interrupt handling or arithmetic exception condition (i.e., overflow, divide by zero, etc.) processing. For certain types of I/O exception conditions, control is available to the user.

c. FORTRAN control mechanisms are separable and the user need not pay for unused capabilities. However, FORTRAN control mechanisms are not "composable" in the sense of G1 and, in fact, the DO statement of the previous FORTRAN standard cannot now be composed in FORTRAN [1, Section 11.6.3] because the condition is now tested before the first iteration.

d. The scope of modifications required to bring FORTRAN into compliance with the various aspects of G1 is discussed point by point in the succeeding paragraphs of this section. The scope of the changes taken as a unit is quite large. Implementation would be greatly complicated. Ad hoc addition of (pseudo) parallel processing, recursion, and exception and interrupt handling does not seem feasible. The facilities must be considered throughout the language design process and impact almost all existing features. For example, parallel processing requires extension of declarators so as to permit definition of controlled accessing and major compiler modification to support such. Recursion necessitates new memory management policies and routines to implement these new policies.

e. Within the context of FORTRAN, enabling the composition of control structures is a formidable task. The problem lies in the statement orientation of FORTRAN. While the DO structure can be defined over a sequence of statements, no general facility exists for defining composite statements. This can be seen in the logical IF, which can cause conditional execution of only a single statement.

2. The GoTo (G2).

a. Degree of compliance: P

b. The GoTo statement is present in FORTRAN. FORTRAN has numeric labels, switches, label variables and label parameters, but no designational expressions. With the exception of label parameters (the GoTo a label parameter is a special form of the RETURN statement [1, Section 15.8]) the GoTo is limited to explicitly specified program labels at the same scope level [1, Section 11.1, 11.2 and 11.3]. FORTRAN imposes no unncessary costs for the presence of GoTo.

c. The only valid use of label parameters seems to be in conjunction with error handling. If other exception-handling facilities are added to the language (as required by G7), this feature can be easily removed without impacting the power of FORTRAN.

3. Conditional Control (G3).

a. Degree of compliance: P

b. FORTRAN has three types of conditional control. An arithmetic IF jumps to one of three labelled statements, depending on the value of an arithmetic expression. The logical IF will execute or not execute a statement based on the value of a Boolean expression. There is no facility in FORTRAN for defining an "ELSE" clause. The computed GoTo offers a FORTRAN user a CASE-like mechanism.

c. Since [1] was written, four new statements pertinent to this discussion have been added to the draft proposal. These are ELSE, ELSEIF, ENDIF and a block IF. This still leaves the non-partitioned logical IF, however. Its deletion would have a very small effect if the ELSE clause was permitted to be empty.

d. A general form of condition (e.g., Zahn's device) could be added to the language without great difficulty. However, this type of construct is at a much higher level than existing control structures and would not blend naturally with the rest of the language.

4. Iterative Control (G4).

a. Degree of compliance: P

b. FORTRAN's iterative control mechanism permits the termination condition only at the head of the loop. It does not restrict the scope of control variables, and it offers as its only termination condition a fixed number of iterations [1, Sections 11.6.3 and 11.6.4]. Entry to an "inactive" DO-loop is permitted only at the head of the loop but, because of the presence of label variables, this restriction is difficult to enforce. It is possible to pass values out of a loop, but the loop control variable itself has an unambiguous value [1, Section 11.6.7].

c. FORTRAN's deficiency of iterative control structures reflects its relatively early design in terms of the development of structured programming theory. The addition of more modern and general control structures requires a moderate amount of effort in both design and implementation. A general loop-while statement analogous to the one proposed by Knuth [12] would bring FORTRAN into greater compliance with this requirement.

d. Limiting the scope of the loop control variable in the DO statement would constitute a relatively minor change in the language specification. It could be accomplished simply by stating that use as a control variable causes an identifier to become undefined after loop exit. Recognition of references to undefined variables has already been discussed in connection with requirement D3.

5. Routines (G5).

a. Degree of compliance: F

b. FORTRAN explicitly forbids recursion: "A subprogram must not reference itself, either directly or indirectly" [1, Section 15.6.2.2].

c. Adding recursion to FORTRAN requires a minor modification from the language design point of view but extensive support within the compiler. Since FORTRAN is not a block-structured language the issues of restricting the lexical embedding of routines in recursive procedures do not apply. The primary difficulty with introduction of

recursion relates to the requirement for dynamic memory management, which it necessitates. Each new activation of such a routine requires memory to be allocated from some space pool for all local variables. Even if the suggested limitations are placed on recursive procedures, the ability to have mutually recursive routines can cause complex and time-consuming memory management to occur.

6. Parallel Processing (G6).

a. Degree of compliance: F

b. FORTRAN has no facilities for parallel processing.

c. The scope of language modifications needed to support parallel processing would be extremely large. In addition, the effort required during implementation would be great, especially if no operating system support may be assumed.

7. Exception Handling (G7).

a. Degree of compliance: F

b. FORTRAN provides the user no capabilities for retaining control in the event of any non-I/O exception, nor is any standard action specified for such exceptions. The I/O exceptions for which a user retains control are largely implementation-dependent [1, Section 12.6.11].

c. The first step to compliance with G7, is the definition of all possible (or recoverable) error conditions. This would require great effort if total generality is desired. The introduction of language features for specifying the error-handling routines would be much simpler; however, they could necessitate extensive compiler and run-time support. In the absence of hardware detection of these exception conditions, an excessive run-time overhead would be incurred.

8. Synchronization and Real Time (G8).

a. Degree of compliance: F

b. The only facility which FORTRAN contains for causing delay on a program path is the PAUSE statement. This simply provides the ability to stop execution until some undefined

operator intervention occurs. This in no way satisfies the expansive requirements of G8.

c. The PAUSE statement could be expanded to include suspension of execution for a fixed amount of time or until some external event (other than operator intervention) occurs. However, the ease with which this could be implemented greatly depends on the system on which the compiler is meant to run. In the absence of system support, this would entail a large amount of work.

d. Assigning relative priorities among parallel control paths and permitting asynchronous hardware interrupts to be treated as any other exception situation are actually added requirements of G6 and G7, respectively. The difficulty of implementation is dependent on how the modifications suggested in these sections are accomplished and the underlying system configuration.

## Section IX. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics (H1).

a. Degree of compliance: P

b. FORTRAN has a "semi-fixed" format: positions 1-5 of a line or card are reserved for a statement label if any is to appear; position 6 is a continuation denoting if the current line or card is a continuation of the previous line or card; positions beyond 72 are ignored; and positions 7-72 contain the FORTRAN statement in a free form. Mnemonically significant identifiers of six or fewer alphanumeric characters may be used. Since blanks are ignored in FORTRAN, lexical analysis is relatively complex. FORTRAN has some context-dependencies and is not easily parsed; on the other hand, there are a number of efficient ad hoc techniques directed especially at parsing FORTRAN so that FORTRAN compilers are generally quite fast. FORTRAN does not allow abbreviations of keywords or identifiers. The FORTRAN assignment statement and FORTRAN expressions are based on conventional forms. The positional significance of DO-statement components and arithmetic IF-statement jump addresses result in a lack of clarity. Because in FORTRAN blanks are not significant and keywords are not reserved, some programmer errors can produce completely unintended but still valid statements (e.g., "DO 315 I=1, 10" executes the DO-loop ten times, whereas "DO 315 I=1.10" assigns 1.10 to a variable DO315I).

c. The changes necessary to permit free format coding of FORTRAN are relatively minor in terms of language design. The additional compiler code for interpreting free-format programming, however, would be reasonably large. Also, from a historical point of view this would constitute a major change in design.

d. Regarding readability, the arithmetic IF can be deleted from the language, since this statement has now been superseded by new conditional constructions. Probably the only reason this has not been done in the new draft proposal is that it would limit upwards compatibility. One should also note here that despite FORTRAN's somewhat unclear statements (e.g., DO) the language is so widely used that most programmers are familiar with their meaning.

2. No Syntax Extensions (H2).

a. Degree of compliance: T

b. The FORTRAN user has no facilities for modifying any aspect of the source language syntax.

3. Source Character Set (H3).

a. Degree of compliance: T

b. The FORTRAN character set consists of (1) the Alpha characters A-Z, (2) the Numeric characters Ø-9, and (3) the special characters blank = + - * / () , . $ ' : [1, Section 3.1], all of which appear in the USASCII 64 character subset.

4. Identifiers and Literals (H4).

a. Degree of compliance: PT

b. FORTRAN provides formation rules for identifiers and literals. Except for those internal to character string literals, FORTRAN ignores the presence of blanks; thus blanks may be used as break characters for identifiers and numeric literals. The maximum number of characters allowed to compose an identifier is six. FORTRAN does not require separate quoting of each line of a long literal.

c. Requiring separate quoting of each line of a long literal is a trivial change to the language.

5. Lexical Units and Lines (H5).

a. Degree of compliance: F

b. FORTRAN allows the continuation of lexical units across lines. There is no way to include end-of-line in literal-strings.

c. The inclusion of an end-of-line notation within literals implies only a very minor change to the language. For consistency with input-output formatting a slash (/) would probably be best. This would necessitate another notation (possibly a double slash) when one wishes to actually include this symbol. Restricting lexical units to a single line requires similarly trivial changes.

6. Keywords (H6).

a. Degree of compliance: P

b. FORTRAN keywords are not reserved, are few in number, and are relatively informative. Since they are not reserved, any keyword may be used as an identifier.

c. Reserving all keywords constitutes an extremely minor language modification. Implementation would be made easier since parsing would be simplified and error recovery could be improved. The ability to define identifiers which are also keywords is seldom used and consititutes poor programming.

7. Comment Conventions (H7).

a. Degree of compliance: P

b. FORTRAN has a single uniform comment convention. Comments may be easily distinguished from code, permit any combination of characters to appear, terminate automatically at end of line, and are introduced with a single language character. FORTRAN comments are introduced with a "C" or an "*" in position 1 of a source line or card [1, Section 3.2.1]. This convention prohibits some reformatting of programs. A major drawback is that a comment may not appear on the same line as the statement to which the comment applies.

c. Required here is a second comment convention which is fully bracketed and may appear internal to statements. The additional work needed in both design and implementation is extremely small.

8. Unmatched Parentheses (H8).

a. Degree of compliance: T

b. Parentheses must be balanced in FORTRAN programs. Every program unit requires an END, and every DO-loop requires a labelled end-of-loop statement.

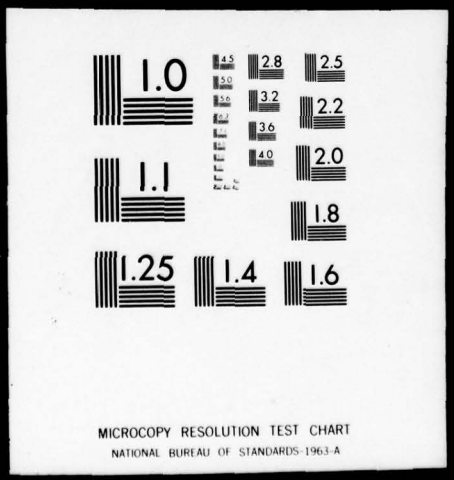
AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

16 OF 28
ADA037634

16 OF 28

ADA037634



9. Uniform Referent Notation (H9).

a. Degree of compliance: P

b. FORTRAN has one referent notation which is valid for function calls [1, Section 15.2.2] and data reference [1, Section 5.3]. The maximum number of dimensions of an array is seven but the maximum number of arguments of a function is not explicitly specified. The fact that function calls cannot appear on the left side of assignment statement is in conflict with the uniform referent requirement.

c. The limit on the number of possible array dimensions seems reasonable and is required to comply with I6.

d. Incorporation of a facility for permitting assignment to functions requires major work in both design and implementation.

10. Consistency of Meaning (H10).

a. Degree of compliance: P

b. FORTRAN has a consistent notation. For instance, "=" is the assignment symbol and ".EQ." is the comparison operator. FORTRAN conventions do imply different interpretations for parenthesized arguments, but that is consistent with the defined syntax and semantics for expressions appearing in argument lists [1, Section 15.9.3].

## Section X. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic (I1).

a. Degree of compliance: F

b. FORTRAN fails to satisfy this requirement; in fact, the standard asserts that whether illegal programs are diagnosed is implementation dependent [1, Section 20.1]:

> What this standard calls a "processor" is any mechanism that can carry out the actions of a program. Commonly this may be any of these:
> (1) The combined actions of a computer (hardware), its operating syatem, a compiler, and a loader.
> (2) An interpeter.
> (3) The mind of a human, perhaps with the help of paper and pencil.
> When you read this standard, it is important to keep its point of view in mind. The standard is written from the point of view of a programmer using the language, and not from the point of view of the implementation of a processor. This point of view affects the way you must interpret the standard. For example, in section 3.3 the assertion is made:
> ".... a statement must not contain more than 1320 characters."
> This means that if a programmer writes a longer statement, then his program is not standard-conforming. Therefore, it will get different treatment on different processors. Some processors will accept the program, and some will not. Some may even seemingly accept the program but process it incorrectly! The assertion means that standard-conforming processors must accept statements up to 1320 characters long. That is the only inference about a standard-conforming processor that can be made from the assertion. The assertion does not mean that a standard-conforming processor is prohibited from accepting longer statements. Accepting longer statements would be an extension.
> The assertion does not mean that a

standard-conforming processor must diagnose statements longer than 1320 characters, although it may do so.
In some places, explicit prohibitions or restrictions are stated, such as the above statement length restriction. Such prohibitions are on what programmers can write in standard-conforming programs. As such they have no more weight in the standard than an omitted feature. For example, there is no mention anywhere in the standard of double precision integers. Because it is omitted, programmers may not use this feature in standard-conforming programs. A standard-conforming processor may or may not provide it or diagnose its use. Thus, an explicit prohibition (such as statements longer than 1320 characters) and an omission (such as double precision integers) are equivalent to this standard.

c. As can be seen from paragraph b, the design philosophy of FORTRAN is in direct opposition to this requirement. For this reason, compliance would entail fairly extensive modification of [1]. The limits set on standard-conforming programs need to be extended to include the translator and the action to be taken when a non-standard form is encountered. This would greatly limit the flexibility of implementation and necessitate more development effort before any compiler could be judged standard-conforming. It also would set a lower limit on the complexity of any compiler. This would increase the minimum size of a FORTRAN translator and thereby prohibit its use on certain machines.

2. Object Representation Specification Optional (I2).

a. Degree of compliance: F

b. The FORTRAN language has no provisions for specifying the object representation of data or programs.

c. The addition of object representation specification facilities would constitute a major redesign of the language and greatly complicate implementation. The impact on existing facilities would be both pervasive and complex. For example, the COMMON and EQUIVALENCE statements would

need to be either restricted to associating variables of identical object representation or drastically changed in nature (possibly totally removed from the language). New rules for parameter passing and operations between variables of different object storage would have to be defined and adhered to by implementations.

3. Compile Time Variables (I3).

a. Degree of compliance: F

b. FORTRAN contains no compile-time variables and has no facilities for their definition.

c. This requirement necessitates relatively minor changes to the language since the addition of compile-time variables does not impact present facilities. The only possible interaction would occur when a programmer attempts to define an identifier with the same name as a compile-time variable. This possible ambiguity can easily be removed by the establishment of a standard interpretation.

d. The extent of needed compile-time support of this feature depends upon the number and complexity of inquiries allowed. If the compiler is to maintain an extremely detailed model of the object machine, the task of both designing and implementing these facilities will greatly increase in difficulty. In addition, in a quickly evolving environment frequent compiler updates would be required. However, a relatively simple model could be developed to include the system parameters required by I3, and it is to such a model that the comments in c apply.

4. Conditional Compilation (I4).

a. Degree of compliance: F

b. FORTRAN contains no conditional compilation facilities.

c. A simple conditional compilation facility could be added to FORTRAN fairly easily. Real difficulties can be encountered, however, if this capability is made overly general and frequently used. For example, if the compilation becomes subtly linked to a particular machine configuration, then small changes in the system could cause

new and hard-to-locate bugs. This is especially true when a program is made installation-dependent (as opposed to machine dependent).

5. Simple Base Language (I5)

a. Degree of compliance: PT

b. The concept of a "base" or "kernel" language is inapplicable to FORTRAN, since FORTRAN is very weak in definitional facilities. Thus the entire language must be considered as the base. Seen in this light, FORTRAN is fairly successful in satisfying I5. The language is relatively simple, and there are few duplicated facilities (examples of the latter are the WRITE and PRINT statements, and logical and arithmetic IF statements). FORTRAN object code can be efficient, and its degree of safety depends on the implementation and whether "dangerous" features (such as EQUIVALENCE) are used. Program understandability depends on whether (and how) features such as label assignment and the COMMON statement are used.

c. The scope of modifications required here depends upon the changes made to the language in conjunction with other requirements. For example, if extension facilities are added, the "base" would need to be extended to include a facility by which these could be defined.

d. As it stands, FORTRAN requires only minor changes to bring it into full compliance. These would be primarily in the direction of pruning the language of redundant features (e.g., the arithmetic IF and statment functions).

6. Translator Restrictions (I6).

a. Degree of compliance: P

b. FORTRAN limits explicitly the number of array dimensions and the length of identifiers but does not limit the levels of nested parenthesis or the number of identifiers.

c. The addition of all required limitations to the language definition is straightforward. The problems lie in making the distinction between machine and translator limits. FORTRAN is probably used on more different machines

than any other language. To expect the FORTRAN used on the CDC STAR to have no more identifiers than some maximum determined for a PDP-8 seems unreasonable.

7. Object Machine Restrictions (I7).

a. Degree of compliance: T

b. FORTRAN has no restrictions which are a result of a specific object machine implementation.

## Section XI. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

1. Efficient Object Code (J1).

a. Degree of compliance: PT

b. FORTRAN does not impose run-time costs for unneeded or unused generality, and compilers for FORTRAN should be capable of producing efficient code for all programs. It may be noted that there has been an exception to this with past versions of FORTRAN with respect to character processing. The definition of character strings given in [1] should allow better code to be produced for programs processing character data. The FORTRAN language specification does not state how features in the language are to be implemented; however, the use of libraries is common and thus only the capabilities used need be available at run time.

c. A formal definition of library facilities could be easily added to [1]. This would, if done properly, bring FORTRAN into full compliance with this requirement. (See our above comments concerning F4.)

2. Optimizations Do Not Change Program Effect (J2).

a. Degree of compliance: P

b. FORTRAN has attempted to meet this requirement by leaving order of evaluation unspecified but restricting the side effects permitted in FUNCTIONs [1, Section 6.6.2]:

> If a statement contains more than one function reference, a processor may evaluate the functions in any order, except for a logical IF statement and a function argument list containing function references...
>
> In a statement that contains more than one function reference, the value provided by each function reference must be independent of the order chosen by the processor for evaluation of the function references.

However, since enforcement of the side-effect restrictions is outside the scope of the language standard, in practice different orders of evaluation may yield different results. In addition, side effects not prohibited by the above rules

can cause problems. For example, assume that F is a function which assigns its argument to the global (COMMON) variable I and then returns this value. Then the expression F(1) + F(2) will yield the same result (3) whichever invocation is evaluated first, but the value of I on completion of the evaluation is dependent on the order in which F(1) and F(2) are performed.

c. The enforcement of side-effect restrictions requires considerable additions to the compiler. Instead of the needed global program analysis, many common FORTRAN optimizations would be forced to be removed from standard-conforming compilers. This could have drastic effects on program efficiency. (These issues were also reviewed in connection with requirement C1.)

3. Machine Language Insertions (J3).

a. Degree of compliance: P

b. FORTRAN permits external procedures to be coded in any language (including machine language) [1, Section 15.6]; however, the responsibility for proper parameter handling is the programmer's. This is not, in fact, as great a difficulty as it sounds, due to FORTRAN's extremely simple parameter passing mechanism. Additionally, FORTRAN does not allow these (or any other) routines to be declared open, thereby limiting their utility in optimization. Since conditional compilation is not available in FORTRAN (although required by I4) these subprograms cannot be encapsulated as suggested.

c. The problems associated with allowing open procedures are small and are discussed below in connection with J5. A conditional compilation capability is fairly easy to implement. (See our discussion of I4 above.) The subroutine mechanism provides most of the needed encapsulation characteristics. Taken as a whole, the few changes required for full compliance are small and implementation would not be greatly complicated.

4. Object Representation Specifications (J4).

a. Degree of compliance: F

b. FORTRAN provides no facilities for specifying object representations of data, nor is there, within the standard, any ability to specify space/time tradeoffs.

c. The modifications needed to support object representation specification are reviewed in connection with I2. The introduction of optimization directives requires only minor changes in the language definition but possible extensive support from the compiler. Due to the simplicity of FORTRAN and the specification within [1] of required storage characteristics for all variables, very few space/time tradeoffs may be made by the translator. However, if the language is modified to permit greater compliance with the Tinman, many such situations could arise.

5. Open and Closed Routine Calls (J5).

a. Degree of compliance: F

b. The FORTRAN user has no ability to specify whether an open or a closed routine is to be used.

c. This requirement necessitates a trivial change to the language definition (i.e., adding the ability to declare subroutines opened or closed). Some changes to the implementation are necessary; however, these would be fairly minor.

## Section XII. PROGRAM ENVIRONMENT

1. Operating System Not Required (K1).

a. Degree of compliance: F

b. While no explicit reference to the existence of an operating system appears in [1], an underlying assumption of one's availability is made.

c. There are many aspects of FORTRAN which in the absence of an OS would require extensive run-time support. The I/O statements in FORTRAN require a file and device management system. Inquiry routines must be supported to return information about the characteristics of these I/O objects. A direct (random) access mechanism must exist. Implementation of a compiler which alone satisfies all the requirements of the present language would be a large task. In addition, if facilities exist to meet the other Tinman directives (e.g., parallel processing, exception and interrupt handling), the job would become even more difficult.

2. Program Assembly (K2).

a. Degree of compliance: U

b. The language definition [1] neither prohibits nor requires a compiler to have the capability to support integration of separately written modules. The EXTERNAL declarator implies a separate definition facility, but this facility is not described.

c. Many installations support the required capability. No feature of the language prohibits or even greatly complicates such a facility. The EXTERNAL statement can be used in conjunction with separately written modules and provides most of the specification abilities required. However, if complete interface checking is to be easily accomplished then this statement should be expanded to include specification of the number and types of parameters. This requires only minor language redesign and small additions to the compiler.

3. Software Development Tools (K3).

a. Degree of compliance: U

b. The family of tools available for a given language does not form part of its definition.

c. Due to FORTRAN's widespread use and relative simplicity many software tools have been developed for it. To supplement FORTRAN's relatively weak structured control facilities many preprocessors have been developed for use with it (e.g., see [13] which catalogs over 50 of them). In addition, many linkers, loaders and debuggers exist for the language.

4. Translator Options (K4).

a. Degree of compliance: U

b. This requirement consists of a set of suggestions for translator options, and explicitly prohibits the language definition from defining any given set of options. [1] makes no reference to possible compiler options.

5. Assertions and Other Optional Specifications (K5).

a. Degree of compliance: P

b. FORTRAN contains no facility (except for comments) for permitting inclusion of assertions, assumptions, axiomatic definitions of data type, debugging specifications, or units of measure.

c. The introduction of new commenting form(s) for these purposes would be trivial.

## Section XIII. TRANSLATORS

1. No Superset Implementations (L1).

   a. Degree of compliance: F

   b. [1] states that "a standard-conforming processor may allow additional forms and relationships provided that such additions do not conflict with the standard forms and relationships." Therefore, FORTRAN totally fails to meet the requirement (i.e., it permits virtually unlimited superset implementations).

   c. Removing permission for superset implementations from [1] and replacing it by an explicit prohibition requires rewriting of only a small portion of the defining document. This, however, precludes the use of such a mechanism for realizing the needed modifications already discussed. If such a mechanism is not used, it implies a lack of upwards compatibility. This would create a great uproar due to the deep entrenchment of present FORTRAN.

2. No Subset Implementations (L2).

   a. Degree of compliance: F

   b. Approximately half of [1] is devoted to the definition of a FORTRAN subset.

   c. The FORTRAN subset was defined to encourage the use of the language in contexts where the compiler for the full language would require too great a development effort or would not fit on the current computer. If the subset definition is removed from [1] (a simple task), FORTRAN could not be used in these situations. The extension of a subset compiler to handle the entire language would be a large job.

3. Low-Cost Translation (L3).

   a. Degree of compliance: T

   b. To the extent that language design can affect translation costs, FORTRAN encourages low-cost compilation. The language is relatively simple, and few real requirements are placed on what the compiler must do.

4. Many Object Machines (L4).

a. Degree of compliance: U

b. This requirement is directed at translator design and has small impact on language definition. It should be noted, however, that FORTRAN is a widely used HOL and has been implemented on more machines than any other language.

5. Self-Hosting Not Required (L5).

a. Degree of compliance: U

b. This requirement applies only to translator design.

6. Translator Checking Required (L6).

a. Degree of compliance: F

b. [1] is an extremely weak standard. Any actions, including none, taken by a translator when supplied with an incorrect (not standard-conforming) program, are valid. This implies that violations of the language syntax, type compatibility rules, or semantics restrictions can go unreported.

c. The modifications to [1] which are required to make it a strong standard are relatively small. The definition of a standard-conforming translator must be changed. This could somewhat complicate implementation of small and fast compilers, since at present they can ignore possible standards violations.

7. Diagnostic Messages (L7).

a. Degree of compliance: FU

b. No set of error and warning situations are suggested in [1]. This being the only language-design-oriented requirement of L7, FORTRAN fails to comply.

c. The definition of error situations and messages would be a fairly small task and could be included in [1]. The implications on compiler implementation of forcing error reporting have been discussed in connection with requirement L6.

8. Translator Internal Structure (L8).

a. Degree of compliance: T

b. The FORTRAN language definition does not dictate the characteristics of translator implementations. The philosophy of [1] can be seen in the following statement: "What this standard calls a 'processor' is any mechanism that can carry out the actions of a program."

9. Self-Implementable Language (L9).

a. Degree of compliance: U

b. The language in which a translator is written is not the province of the language definition.

c. The difficulties with writing a compiler in FORTRAN stem from its inability to easily define the necessary data structures and to interface cleanly and efficiently with the object machine.

## Section XIV. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only (M1).

a. Degree of compliance: T

b. FORTRAN, being one of the earliest HOLs, has been thoroughly tested in practical applications. The new constructs and facilities added by the draft proposal [1] are all well within the current state of the art.

2. Unambiguous Definition (M2).

a. Degree of compliance: P

b. The draft FORTRAN proposal was written with emphasis upon readability. The document is directed towards use by programmers and presents only an informal English specification of language semantics. Ambiguities have been avoided, however, and a formal "railroad track" definition of syntax is presented as an appendix to the report. The major drawback of [1] is that it defines only the operation of a standard-conforming program and allows the reader to infer the translator requirements. Formal specification of the language (e.g., in VDL) does not form part of the defining document. Because of its idiosyncracies and many storage-oriented constructs, FORTRAN is not readily amenable to formal definition.

3. Language Documentation Required (M3).

a. Degree of compliance: P

b. FORTRAN syntax is presented for each statement in an easily-understood metalanguage, and the corresponding semantics are given in English. The language is fairly easy to learn, and the defining document is directed at potential users. The action of any legal program cannot be determined from the program and the language definition due to implementation-dependent I/O statements, character collating sequence, and variable range and precisions.

c. Discussion of these implementation-dependent language features occurs earlier in connection with requirements B10 for I/O, A5 for collating sequences, and A3 and A4 for precision and range.

4. Control Agent Required (M4).

a. Degree of compliance: U

b. Management of FORTRAN is outside the scope of the defining document. One can, however, foresee trouble in any attempt to enforce DoD standards on this language. Due to the widespread use and large industrial investment in FORTRAN, any attempt to control the language and its translators, especially if drastic modifications are planned, will meet resistance.

5. Support Agent Required (M5).

a. Degree of compliance: U

b. Identification of support agents responsible for maintaining the translators and language aids is not within the province of the language definition. Support on a DoD-wise basis should be unaffected by the language chosen.

6. Library Standards and Support Required (M6).

a. Degree of compliance: U

b. Maintenance of common libraries is outside the province of language definition. Some difficulty will arise in this area due to the proliferation of already-existing FORTRAN libraries; however, DoD-wide control and support should be realizable. The comments made above in connection with requirement M4 apply equally well to any attempt to fully control FORTRAN libraries.

## Section XV. CONCLUSIONS REGARDING FORTRAN

1. Conflicts between Objectives of FORTRAN and Tinman.

a. Of major consideration in the drafting of [1] was compatibility with the earlier FORTRAN standard. This was inevitable, due to the large investment already existing in FORTRAN software. It implies, however, that the language is still locked into the original design concepts. The last FORTRAN standard, ANS X3.9-1966, was published in 1966, and the theory of language design has evolved greatly since. An increasing emphasis on readability, reliability, and abstraction techniques has occurred, and this new emphasis is visible to only a very limited extent in [1].

b. In FORTRAN's design, portability is secondary to availability (i.e., ease of implementation). Evidence of this exists in the weakness with which limitations are placed on translators for the language. In the case of incorrect programs, the translator is empowered to take whatever actions (including none) that it sees fit. This means that possible semantic violations need not even be checked. To further the use of FORTRAN in differing environments, the standard (1) permits language extensions, (2) suggests subset implementations, and (3) leaves undecided many issues that could possibly complicate implementation (e.g., relative ordering of characters and digits, character vs. non-character storage requirements).

c. The Tinman, on the other hand, reflects the present trend in language design. The major emphasis of that document is upon program reliability and readability and upon language power. The goal of efficiency is seen to be achievable through comprehensive user specification facilities and intelligent compilation as opposed to language simplicity.

2. Summary of Major Areas of Conflict Between FORTRAN and the Tinman.

a. Data and Types. An indication of the differing viewpoints can be seen in the limited facilities FORTRAN offers in this area. FORTRAN contains no fixed-point data type besides integers, no record definition facilities, limited and implementation-dependent precision specification and only static memory allocation (e.g., compile-time

determined array bounds). Additionally, while type-compatibility for actual and dummy parameters is required, the translator may optionally omit type checking. Also, the COMMON and EQUIVALENCE statements effectively disable type checking by permitting free union.

b. Operations. The primary conflicts between FORTRAN and the Tinman in this area are the language's failure to support enumeration types, power sets, range specifications, and operations on data aggregates (i.e., arrays). In addition, FORTRAN supports mixed-mode arithmetic with implicit type conversions and only bitwise equality for floating-point numbers.

c. Expressions and Parameters. In this area, conflicts stem from FORTRAN's only partially defined order of evaluation, its lack of dimensionality constraints on array parameters, and its failure to support generic procedures and variable numbers of parameters.

d. Variables, Literals, and Constants. FORTRAN's lack of step size and range specifiers, separate variable initialization and declaration statements, and lack of pointer variables are its major deficiencies in this area.

e. Definition Facilities. This is the area where FORTRAN deviates most from the Tinman's requirements. There are no facilities for user definition of data types. The only data definition available is the array. For this reason the issues of encapsulated type definitions and operator extensions do not even apply. In addition, FORTRAN permits free union through the EQUIVALENCE statement.

f. Scopes and Libraries. While [1] does not explicitly define library and compool facilities, their inclusion within the language is relatively simple. The EXTERNAL declarator implies the existence of such a feature.

g. Control Structures. FORTRAN provides only a limited control structure. The degree of effort necessary to force FORTRAN into compliance in this area seems prohibitive. The language contains no parallel processing, no exception- or interrupt-handling facilities, no recursion, and no facilities for composing control structures. In addition, the logical IF is not fully partitioned.

h. Syntax and Comment Conventions. FORTRAN is a semi-fixed format language. No separate quoting is required for multi-line literals, and lexical units may continue across lines. Keywords are not reserved and only full-line comments are available.

i. Defaults, Conditional Compilation, and Language Restrictions. FORTRAN seriously conflicts with the Tinman philosophy of no defaults. The most obvious default is that relating the first letter of a variable with a given type. Possibly more crucial, hovevever, is the implementation-dependent range and precision associated with numeric variables and the only partially-defined character collating sequence. In addition, FORTRAN lacks compile-time variables and facilities for conditional compilation and object representation specification.

j. Efficient Object Representations and Machine Dependencies. Efficiency is achieved in FORTRAN through simple language constructs and permissive optimization policies. This results in optimizations which can change the effect of the computations. The language includes no space/time tradeoff directives and no open procedure specifiers.

k. Program Environment. The program environment requirements are for the most part beyond the province of language definition. To the extent that [1] impacts on these requirements, only small differences exist.

l. Translators. A basic disagreement of philosophies exists in this area. The Tinman requires a strict language standard allowing no superset of subset implementations, whereas [1] defines a compatible subset and permits unlimited supersetting.

m. Language Definition, Standards and Control. In terms of language definition, FORTRAN fares well. [1] is extremely readable, although arbitrary implementation dependencies led to only partial semantic definition of several language features. Control of FORTRAN can be seen to be difficult. The widespread use and large amounts of software already existing in the language led to many groups with vested interests in controlling the language.

3. Unnecessary Features of FORTRAN

a. Several features of FORTRAN are neither prohibited nor required by the Tinman. The arithmetic IF, ASSIGN, assigned GOTO and statement function statements are all relics of earlier versions of FORTRAN which fall into this category. The mechanism supplied for creating label variables and jumping to locations thereby specified conflicts with the Tinman philosophy of reliability. Since label variables are declared to be of type integer, this entails a violation of type rules. In addition, the availability of label variables prohibits compile-time enforcement of control transfer rules (e.g., jumping into the middle of a DO loop). The arithmetic IF and statement function statements are now redundant due to the introduction of the logical IF and FUNCTION subprograms, respectively.

b. The ENTRY statement which has been introduced into FORTRAN by the new draft proposal is another feature not impacted by Tinman requirements. However, it provides a unique facility in FORTRAN and should not be removed. Along with the SAVE declarator it permits limited access data structures to be defined.

4. Recommendations concerning FORTRAN.

a. We began this section by discussing the basic differences of philosophy between FORTRAN and the Tinman. Throughout this chapter, we have seen the many fundamental modifications that are needed for FORTRAN's compliance with the Tinman. In addition, this section has emphasized the expected resistance to any major changes in the language. For these reasons, it is felt that the choice of FORTRAN as a base for development of a Tinman-like language would be ill-advised.

b. The potential advantages of selecting a language already in widespread use are obvious (availability of programmers, translators, support tools and documentation). However, the changes required in FORTRAN are of such a global nature as to eliminate these potential gains. The necessary modifications would drastically alter the language necessitating retraining of programmers, and rewriting of existing software and documentation.

c. In summary, FORTRAN as proposed in [1] is incompatible with the Tinman, and modification to achieve compliance would be prohibitive and destroy the existing "flavor" of the language.

# CHAPTER 7

# COBOL EVALUATION

## Section I. LANGUAGE SUMMARY

### 1. Lexical Properties.

COBOL is primarily card-oriented rather than free-format, as illustrated by the rules concerning the placement of continuation lines and division and section headers. Spaces are significant in separating lexical units. Identifiers may be up to thirty characters long, and the hyphen serves as a break character. There are a large number of reserved words (about 300), including some with alternate spellings and abbreviations. An asterisk in the continuation indicator area is used to denote comment lines.

### 2. Data Types.

a. The data types available in COBOL may be categorized according to Figure 6.

b. Except for Index data, each elementary data-item is considered to be a fixed-length string of characters of various kinds. A numeric item may contain any of the digits ('0' through '9') with a possible sign. An alphabetic item may contain any of the letters or the space character. An alphanumeric item may contain any character in the computer's character set. Numeric, alphabetic, and alphanumeric are called "classes" in COBOL and correspond roughly to data types; however, run-time checks must be made to guarantee that data declared as numeric or alphabetic in fact do not contain improper characters. It should be pointed out that the central role of the character string in COBOL's data scheme is not surprising, in view of the heavy orientation of that language toward the I/O requirements of business data processing.

c. As shown in Figure 6, numeric data are of two varieties. The condition data facility provides a limited form of enumeration type, allowing a symbolic name to be used (but only in conditional statements) in place of a test involving an explicit numeric value. We use the term "standard" numeric to refer to numeric items which are not defined as condition data.

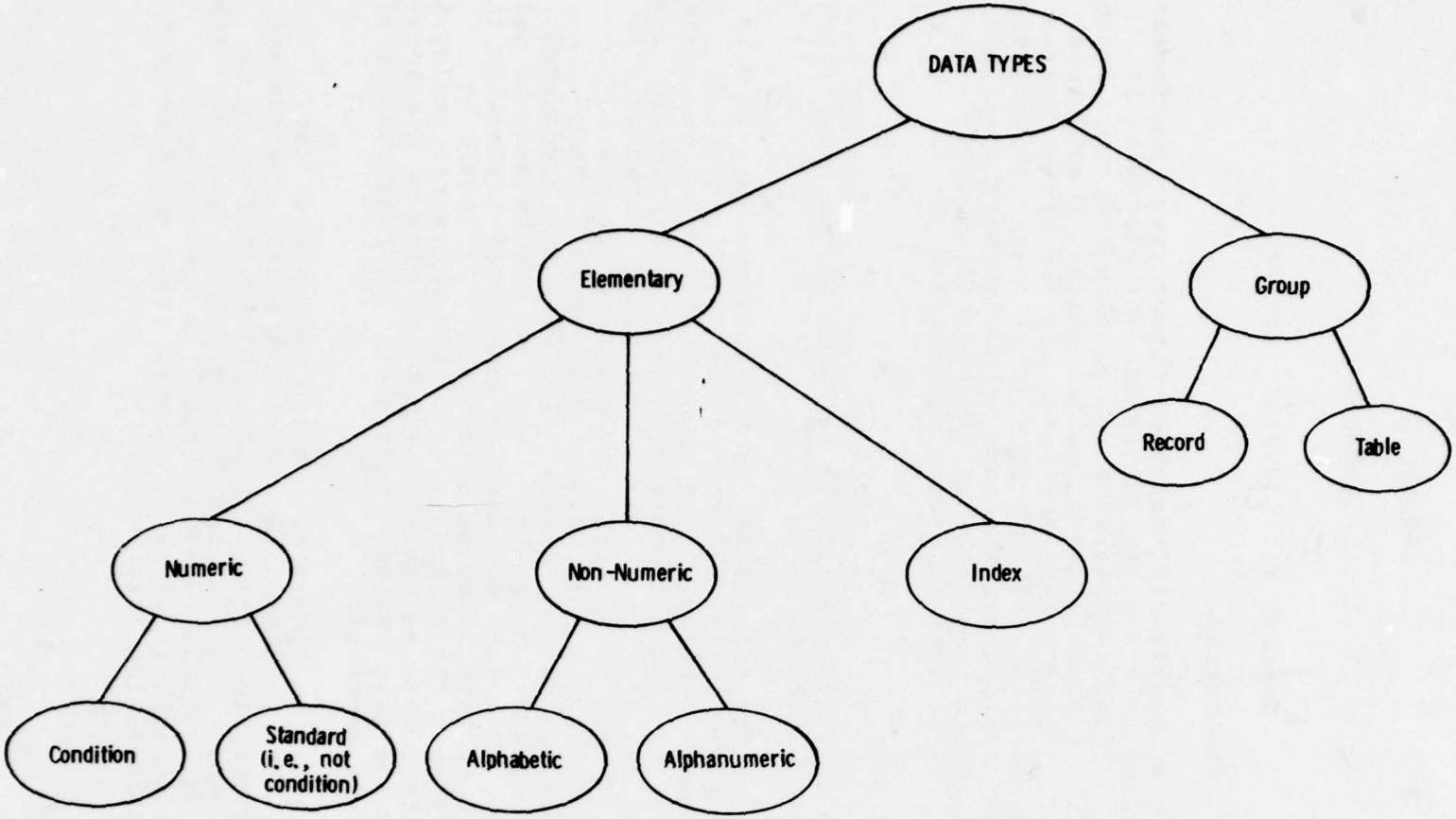



Figure 6. Data Types in COBOL

d. An Index data-item has special use in table handling. It is provided for efficient subscripting on the object machine.

e. Group data-items may be records (hierarchically composed structures) or tables (one-, two-, or three-dimensional arrays). Independent of the classes of items, a group is regarded as a string of alphanumeric characters for the purpose of using it as a complete entity (e.g., in MOVEs).

f. Overlaying of storage is allowed in COBOL via the REDEFINES and RENAMES clauses.

3. Procedures.

COBOL's subroutine facility is quite weak. "Function" subroutines (i.e., routines which return values) are not provided. In order to define a routine which takes parameters, that routine must be embodied in the PROCEDURE DIVISION of a separate program; moreover, the only kind of parameter passing allowed is "by reference," and no type checking is performed. Within a single program, the only kind of routine which may be defined is one which takes no parameters and which has access to all the data in the DATA DIVISION.

4. Statements.

a. Arithmetic Statements. The ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE statements supply COBOL's arithmetic capabilities.

b. Assignment. Assignment in COBOL can be carried out via any of the arithmetic statements or through the MOVE statement. Implicit conversions (from display to computational format or vice versa) are performed.

c. GoTo. COBOL's GO TO statement allows branching to any paragraph (or section) in the PROCEDURE DIVISION. Dynamic modification of the target of a GOTO is permitted via the ALTER statement.

d. Conditional Statements. COBOL provides an IF statement, with both THEN and ELSE clauses mandatory (except when the statement is at the end of a sentence, in which

case the ELSE clause may be omitted). The GO TO ... DEPENDING statement serves as a restricted form of the "case" construct.

e. Iteration. The versatile PERFORM statement serves both to invoke a procedure and to carry out iterated loops. In the latter capacity, the PERFORM statement has options permitting a fixed number of iterations as well as loop repetition until a Boolean condition (tested at the beginning of each iteration) becomes true. The specification of a loop with a test for completion in the middle is awkward.

f. Miscellaneous. A variety of special purpose statements is supplied by COBOL. Examples are the Nucleus Module's INSPECT statement, which permits tallying and/or replacement of characters in a data item, and the Sort-Merge module's SORT and MERGE statements.

5. Storage Allocation.

COBOL provides for static allocation of data; i.e., the sizes of all records and data-items are known at compile-time. (With "varying length" tables, the maximum size is known at compile-time.) The only exception to this occurs when an externally defined routine is invoked (i.e., using the Inter-Program Communication Module); in such a case the storage for the called routine is (implicitly) allocated dynamically, and may be (explicitly) deallocated under program control.

6. Process Scheduling.

COBOL contains no facilities for process scheduling.

7. Files and I/O.

The variety of I/O facilities provided by COBOL is the language's main strength. The following paragraphs, from [8, pp. XIV-38, 39], summarize the mechanisms available.

a. Sequential Organization. "A file whose organization is sequential can only be accessed in the sequential mode. Records in such a file can be accessed in the sequence established as a result of writing the records to the file. A sequential mass storage file may be used for input and

output at the same time. One file maintenance method made possible by this facility is to read a record, process it, and, if it is updated, return it, modified, to its previous position."

b. Relative Organization.

(1) "A file whose organization is relative can be accessed either sequentialy, dynamically, or randomly. Sequential access provides the same results as if the file were organized sequentially. Random access allows the sequence in which the records are accessed to be controlled by the programmer. Each record in a relative file is identified by an integer value greater than zero which specifies the record's logical ordinal position in the file. The desired record is accessed by placing its relative record number in a Relative Key data item. Such a file may be thought of as a serial string of areas, each capable of holding a logical record. Each of these areas is denominated by a relative record number. Records are stored and retrieved based on this number. For example, the tenth record is the one addressed by relative record number 10 and is in the tenth record area, whether or not records have been written in the first through the ninth record areas."

(2) "In the dynamic access mode, the programmer may change at will from sequential access to random access using appropriate forms of input-output statements."

c. Indexed Organization.

(1) "A file whose organization is indexed can be accessed either sequentially, dynamically, or randomly. Sequential access provides access to the records in the ascending order of the record key values. The order of retrieval of records within a set of records having duplicate record key values is the order in which the records were written into the set."

(2) "In the random access mode, the sequence in which

records are accessed is controlled by the programmer. Each record in the file is identified by the value of one or more keys within that record, and the desired record is accessed by placing the value of its record key in a record key data item before accessing the record."

(3) "In the dynamic access mode, the programmer may change at will from the sequential access to random access by using appropriate forms of input-output statements."

8. Exception Handling.

COBOL offers a limited set of features for handling exception conditions arising on arithmetic overflow, I/O, and storage bounds overflow. A user-supplied SIZE ERROR phrase can be included in arithmetic statements to be invoked on overflow (including zero-divide). A FILE STATUS data item, AT END phrase, and USE statement are constructs which support handling of I/O exceptions. An ON OVERFLOW phrase allows user control when storage overflow occurs. Further details concerning these features are supplied below, in connection with the discussion of paragraph G7.

9. Compile-Time Facilities.

a. COBOL's COPY statement provides a limited macro facility, allowing the insertion of source text (e.g., commonly-used file or record descriptions) into a program.

b. The facilities of the Debug Module allow the user to monitor, during program execution, the performance of user-selected procedures and the values of user-specified data items.

## Section II. DATA AND TYPES

1. Typed Language (A1).

a. Degree of compliance: FP

b. COBOL satisfies this requirement only to a limited extent. In fact, the notion of "data type" does not readily apply to COBOL. Nearly all elementary data-items, and group data in certain contexts, are regarded as character strings. Properties which are compile-time checked and part of the data type in other languages require run-time checking (sometimes programmer-supplied) in COBOL. For example, it is possible to declare a data-item with a numeric class (77 TEMP PICTURE 999 VALUE ZERO.), but this does not prevent the item from containing non-numeric data (e.g., as inserted during a READ). If the programmer foresees such a possibility, he must provide a validation routine to ensure that the data match the specified picture. For example:
INSPECT TEMP REPLACING LEADING SPACES BY ZEROES.
IF TEMP NOT NUMERIC GO TO DATA-ERROR.

c. It would be a major change, both to the language and implementation, to bring COBOL into compliance with A1. The effort involved would amount to a redesign of much of the language.

2. Data Types (A2).

a. Degree of compliance: P

b. COBOL supplies most of the types required in A2. Integer and fixed-point (up to 18 digits) are combined in the numeric class in COBOL, and character corresponds to alphanumeric. There is no explicit mention of floating-point in the COBOL document. The discussion of the USAGE clause [8, p. II-35] seems to imply a single computational representation (viz., fixed-point), which would rule out floating-point; however, the description of arithmetic expressions [8, pp. II-39-40) leaves to the implementor the treatment of arithmetic expressions, which appears to allow floating-point as a possible implementation technique. COBOL does not contain an explicit Boolean data type but does allow Boolean operations on relationals to appear in conditional statements. Arrays (up to three dimensions) and records are provided.

c. The addition of floating-point is a minor change to the language but requires a moderate amount of implementation effort. Introducing a Boolean type presents problems because of the special rules which would be required to describe the external format of Boolean data. The specification of arrays and records as type generators would be a major change and require substantial redesign.

3. Precision (A3).

a. Degree of compliance: F

b. Lacking a language-defined floating-point facility, COBOL fails to meet this requirement.

c. As mentioned in connection with A2, the addition of floating point is a minor change to the language and has moderate impact on implementation. Providing global (as opposed to data-item by data-item) precision specification would not be difficult, but it would be in direct conflict with the language's conventions concerning fixed-point data.

4. Fixed Point Numbers (A4).

a. Degree of compliance: F

b. COBOL satisfies this requirement completely via its NUMERIC class data.

5. Character Data (A5).

a. Degree of compliance: P

b. Although COBOL lacks a general facility for defining enumeration types, the language does give the programmer the ability to define character sets in a manner similar to enumeration types. In the OBJECT-COMPUTER paragraph of the ENVIRONMENT-DIVISION, the programmer can specify an alphabet-name to be used as the PROGRAM COLLATING SEQUENCE [8, p. II-6]; the description of the collating sequence (e.g., ASCII, or NATIVE to the object computer, or user-defined) appears in the SPECIAL-NAMES paragraph.

c. The changes needed to enable definition of enumeration types in COBOL are described below in connection with E6. It should be pointed out, though, that the

character data type is unlikely to be cleanly obtained using such a facility. For COBOL many of the problems of defining and using enumeration types are more severe than for other languages, since all data attributes in COBOL are described with PICTURE and USAGE clauses. Finding an appropriate format for representing an enumeration type, descriptions of strings of enumeration types and of literal strings of enumeration types are tasks of great difficulty.

6. Arrays (A6).

a. Degree of compliance: P

b. COBOL partially meets this requirement in its Table Handling module. The number of dimensions is restricted (one, two or three), the value of the lower bound is always 1, and the value of the upper bound is always specified as a literal value. COBOL does provide a limited facility for defining "flexible" arrays -- the phrase "OCCURS integer-1 TO integer-2 TIMES DEPENDING ON data-name-1" [8, p. III-2] -- but the programmer must maintain the current upper bound in data-name-1. An annoyance in COBOL is the inability to define tables at the "top level." If one wishes to define TAB1 as an array of three digit integers, one must embed TAB1 in a record to do so -- e.g.,

```
Ø1 TAB1-RECORD
   Ø2 TAB1 OCCURS 2Ø TIMES
           PICTURE 9(3).
```

c. Some of the changes required to comply fully with A6 would alter the basic structure of COBOL. The requirement that "The upper subscript bound will be determinable at entry to the array allocation scope" implies block structure which is absent from COBOL. There are no enumeration types available to use as subscripts. Both of these represent major changes to COBOL.

7. Records (A7).

a. Degree of compliance: P

b. COBOL provides several facilities for defining records with alternative structures, but in each the responsibility is placed on the programmer for checking at run-time which alternative applies. Thus the dangers and the possibility for defeating type-checking are inherent in COBOL.

c. Some COBOL features which specify overlaying are:

(1) the definition of multiple records associated with the same file in the FILE-SECTION of the DATA-DIVISION (the overlaying here is <u>implicit</u> and a possible source of programmer error);

(2) the REDEFINES clause in a record description, which allows the same storage to be specified by different data description entries [8, p. II-27];

(3) the RENAMES clause, which allows alternative, possibly overlapping,, groups of elementary items [8, p. II-29];

(4) the SAME AREA clause, which permits the same storage area to be used for records from more than one file (assuming that no two of these files can ever be open simultaneously).

d. As is frequent in COBOL, there are a fairly extensive set of rules and special cases governing the use of these facilities. For example, it is illegal [8, p. II-27] to REDEFINE a storage area whose description includes an OCCURS clause (i.e., a table), but a table is allowed to REDEFINE another storage area. Also, the reference manual does not define the semantics of REDEFINEing a field which includes an INDEX item -- i.e., rules require that the same number of character positions be present in both the overlaying and overlaid data, but the format of INDEX items is implementation-dependent.

e. In order to comply with A7, COBOL would need a different data description structure allowing the programmer to indicate more specifically which portions of records were related and which condition discriminates between alternate forms of a record. To detect any violations in the input data would involve some difficult implementation problems. COBOL completely lacks the CASE-like construct used in other languages which facilitates processing of records with alternate forms. Provision of all necessary features and imposition of the necessary security for meeting this Tinman requirement would be large tasks.

## Section III. OPERATIONS

1. Assignment and Reference (B1).

a. Degree of compliance: P

b. COBOL partially satisifies this requirement. The assignment operation in COBOL is distributed among a variety of verbs -- e.g., the arithmetic verbs with the GIVING option -- but the principal means of assignment is the MOVE statement [8, p. II-74ff]. The following points in connection with the MOVE statement show the differences between COBOL and the Tinman requirement:

(1) One cannot directly MOVE tables in COBOL; instead, a table must be defined subordinate to a non-table group.

(2) It is not permitted to MOVE a numeric edited data item to another numeric edited data item.

(The above points violate the Tinman requirement that "the assignment operation will permit any value of a given type to be assigned to a variable...of that type....")

(3) Lacking encapsulated type definitions, COBOL does not permit user definition of assignment or access operations.

(4) The MOVEing of group data-items ignores the types of the constituents of the groups, since groups are interpreted as alphanumeric independent of their component items.

(5) The MOVE will in general involve implicit conversions (e.g., numeric to numeric edited). Also, see B8 below.

(6) An INDEX data item is not permitted as either the source or the target of a MOVE.

c. An INDEX data item cannot be initialized with a VALUE-clause [8, p. II-11], nor can an INDEX data item be assigned or referenced using the usual COBOL verbs. An INDEX data item may be assigned only with SET [8,

p. III-11ff], which is reserved for this purpose; an INDEX data item may be referenced in relational tests [8, p. III-6] and table references [8, p. I-89ff]; it may not appear in an arithmetic expression [8, p. II-39].

d. As demonstrated by the rules governing assignment and reference, COBOL and the Tinman were designed with entirely different objectives. Although the class and category of a data name may appear analogous to the Tinman "type," the PICTURE also plays a role in that items in the same class and category with different PICTUREs will require conversion during a MOVE or other assignment. (This accounts for some of the behavior above.) To change this characteristic of COBOL would require comprehensive redefinition of the data description and the internal data organization; the product of such a redefinition would not be recognizable as COBOL.

2. Equivalence (B2).

a. Degree of compliance: P

b. The EQUAL relation in COBOL can be used to compare objects for identity, but the rules provided for its use [8, p. II-41ff] have serious conflicts with the Tinman requirement. First, EQUAL is more general than B2 allows: in COBOL, objects of different data types can be compared for equivalence, with implicit conversions taking place. Second, equivalence of group items takes place by (effectively) a bitwise comparison, independent of the data types of the constituents.

c. The comments under paragraph B7 above apply to this Tinman requirement.

3. Relationals (B3).

a. Degree of compliance: P

b. COBOL permits relational operations (GREATER, LESS, NOT GREATER, NOT LESS) in the same contexts as equivalence; the character collating sequence is used to determine the ordering. COBOL lacks a general enumeration types, and it is not possible to inhibit the definition of the ordering relations.

c. See paragraph E6 for the modifications required for the inclusion in COBOL of enumeration types. It is not possible to define unordered sets in COBOL, as all data are represented by character strings which have the implementation dependent (or ASCII or other) ordering. Inclusion of some method for defining unordered sets would be a large change to COBOL and would be outside the spirit of the language.

4. Arithmetic Operations (B4).

a. Degree of compliance: PT

b. COBOL provides two facilities for performing arithmetic: a set of verbs (ADD, SUBTRACT, MULTIPLY, DIVIDE), and a number of operations (+, -, *, /, **) used in the COMPUTE statement. With the exception that division with a real result is not part of the language (but may be left to the implementor), the requirements in B4 are met. Two points pertaining to the DIVIDE statement are worth making here. First, a format is available which computes the remainder of a division. Second, the DIVIDE...INTO format is sometimes counter-intuitive; e.g., DIVIDE TOTAL-AMOUNT INTO 2 GIVING HALF-SHARE-AMOUNT, despite the fact that it reads as though HALF-SHARE-AMOUNT will receive the value TOTAL-AMOUNT/2, in fact will assign the reciprocal of this value to HALF-SHARE-AMOUNT.

c. The scope of changes necessary to add a floating point data type to the language is presented under paragraph A2 above.

5. Truncation and Rounding (B5).

a. Degree of compliance: P

b. COBOL allows implicit truncation of high-order character positions during assignment [8, p. I-86], contrary to B5. If the user specifies a ROUNDED phrase [8, p. II-50], then rounding as opposed to truncation will occur at the low-order end. If high order truncation occurs and the user has supplied an ON SIZE ERROR clause, a user specified statement will be executed [8, p. II-50].

c. An ON SIZE ERROR clause could be required with all arithmetic verbs. This would be an easily defined and

implemented change; however, it would degrade the efficiency of the program and thus conflict with requirement J1.

6. Boolean Operations (B6).

a. Degree of compliance: P

b. COBOL provides operations for "and," "or," and "not," but not "nor" [8, p. II-45ff]. No short-circuit mode of evaluation exists. One interesting feature of COBOL is the provision for abbreviating Boolean expressions. For example, a EQUAL b OR c is short for (a EQUAL b) OR (a EQUAL c). Also, a condition name can be used by itself instead of an explicit test of a condition variable vs. the value of the condition name.

c. As Boolean expressions are allowed only in limited contexts (IF-statements) a requirement that they be evaluated in short-circuit mode would have only a small impact on the language and an implementation.

7. Scalar Operations (B7).

a. Degree of compliance: P

b. The only provision in COBOL for performing scalar operations or assignment on "conformable" aggregate data is the CORRESPONDING phrase [8, p. II-51] for the MOVE, ADD, and SUBTRACT verbs. However, the correspondence defined in COBOL is dependent on the names of the record components and does not permit operations such as adding a numeric item to an array of such items. The MOVE verb in general allows assignment of any two records, independent of their logical structure; the result is an alphanumeric string copy from the source record to the destination.

c. Restricting arithmetic and MOVE operations to identically structured records would not be a large change to COBOL. However, the inclusion of vector and array operations would be a substantial and undesirable extension to COBOL. The change would be large because tables are embedded in records and because the actions of all operators would need to be extended or changed. Such changes would be undesirable because COBOL does not attempt to be an efficient algebraic language and it has quite different goals from those implied by B7.

8. Type Conversion (B8).

a. Degree of compliance: P

b. COBOL fails to meet this requirement; in fact, implicit conversions (from character code to computational and vice versa) are a fundamental aspect of the language (e.g., in comparisons and assignments). No explicit conversion operations are provided.

c. The considerations involved in B8 illustrate again the differences between the objectives of COBOL and the goals of a Tinman-like language. For example, consider the following declarations:

```
03 DISPLAY-INTEGER PICTURE 999 USAGE IS DISPLAY.
03 COMPUTATIONAL-INTEGER PICTURE 999
     USAGE IS COMPUTATIONAL.
```

Both variables have a range 0 to 999. The "representational ideal" for both variables is three numeric characters; the announcement USAGE IS COMPUTATIONAL has the purpose of allowing an implementation to select a more efficient representation, if one exists, for computation. Whether to use more than one representation and the representations selected are implementor design decisions.

d. The elimination of implicit conversions would have a major impact upon the language and its implementations.

9. Changes in Numeric Representation (B9).

a. Degree of compliance: PT

b. COBOL partially complies with this requirement. Numeric ranges are implicitly determined by the PICTURE clause (e.g., PICTURE 9(4) denotes the range 0 through 9999), and no explicit conversion operations are required between ranges. The run-time exception on truncation will be generated if the ON SIZE ERROR clause is provided; this is possible for the COMPUTE and other arithmetic verbs, but not for the MOVE statement, which could (but need not) generate a compile-time error.

c. For full compliance with this requirement, the following two minor changes are possible:

(1) The ON SIZE ERROR clause must appear with all arithmetic statements (see paragraph B5 above).

(2) Any MOVE which could cause high order truncation must be announced at compile time (an ON SIZE ERROR could be required for such MOVEs).

10. I/O Operations (B10).

a. Degree of compliance: PT

b. One of the most striking facets of COBOL is its versatility in the I/O area. The format of a COBOL program reflects the orientation toward file handling, with the four separate divisions and the partitioning of the DATA DIVISION into a FILE SECTION and other sections. Also, COBOL very nicely separates the physical description of a file (which includes information such as label handling and blocking factor) from the logical description of the records comprising the file. The diversity of I/O which can be programmed (e.g., sequential, relative and indexed I/O) distinguishes COBOL from most other high-order languages.

c. However, there are several aspects of B10 which are not satisfied in COBOL. One is the ability to assign and reassign devices dynamically; COBOL requires establishing statically the connection between internal and external file names (this is done in the FILE-CONTROL paragraph in the INPUT-OUTPUT SECTION of the ENVIRONMENT DIVISION). Another area in which COBOL fails to satisfy B10 is in that language's inability to permit user definition of I/O operations. The Tinman suggests that the definitional facilities of the language (e.g., for data types and generic procedures) should be powerful enough to permit the specification of unique equipment and their I/O operations. COBOL completely lacks such facilities; thus, any I/O handling must be built into the language.

d. It should also be pointed out that COBOL contains no provision for binary I/O, which is consistent with the fact that COBOL lacks binary data.

e. Full compliance with B10 would require the entire restructuring of the COBOL/operating system interface and the complete redefinition of the FILE SECTION syntax and semantics. These are major changes.

11. Power Set Operations (B11).

a. Degree of compliance: F

b. COBOL completely lacks such operations; moreover, there is no bitstring data type in the language and no way for a user to represent bitstrings efficiently or define operations such as union, intersection, and complement.

c. An appropriate declaration form for power sets would be extremely difficult to devise within the framework of the COBOL DATA DIVISION (similar problems exist for enumeration types; see paragraph E6 below). Because COBOL has no definition capabilities any power set operations would have to be included in some level of the base language.

## Section IV. EXPRESSIONS AND PARAMETERS

1. Side Effects (C1).

a. Degree of compliance: P

b. Strictly speaking, COBOL fails to satisfy the requirements of C1. In the arithmetic expression A/B + (C/D) the parentheses would force the evaluation of the quotient C/D first, followed by the evaluation of A/B, according to the rules on [8, pp. II-39 and II-40]. Thus, if each division would result in overflow the side-effects would not be carried out from left to right.

c. Two points are worth noting in connection with this issue. First, since COBOL does not have value-returning functions in the sense of languages like FORTRAN, ALGOL and PASCAL, there is no possibility for the evaluation of an expression's constituent to result in the performance of I/O or the modification of a variable. Thus, the kinds of side-effects which can occur during expression evaluation are extremely limited -- errors from arithmetic overflow or out-of-bounds subscripts appear to be the only ones possible. However, this means that no matter which order of evaluation is used the result of an expression will always be the same. The language description could be simplified (and better object code produced) if the order of evaluation were left undefined.

d. A second point concerns the apparent confusion in the COBOL standard concerning the parsing of expressions (i.e., the decomposition into constituent sub-expressions) as opposed to the order in which the operands of an operator are evaluated. The failure to distinguish between these concepts is evidenced in the language requirement for parentheses (which should only be used to guide the parsing) to determine order of evaluation. To see that the concepts of expression parsing and evaluation order are independent, consider the expression (A+B)/(C+D). The parentheses indicate that the operands to the division operator are the sum A+B and the sum C+D; such an operator-operand structure is independent of whether A+B is evaluated before or after C+D.

e. The changes needed to bring COBOL into compliance with C1 are minor with respect to the language and moderate with respect to implementation.

2. Operand Structure (C2).

a. Degree of compliance: P

b. COBOL partially satisfies the requirements of C2; areas of divergence from the Tinman are the following.

(1) Unary plus and minus take precedence over exponentiation; in COBOL, -5**2 has the value 25, whereas more commonly (e.g., in FORTRAN and PL/I as well as by standard algebraic conventions) the value would be -(5**2), i.e., -25.

(2) Exponentiation is "left-associative" in COBOL: I**J**K is interpreted as (I**J)**K. Again, standard algebraic conventions dictate a different parsing, viz., I**(J**K).

(3) The ability to abbreviate combined relation conditions in COBOL [8, p. II-47-48], though a possible convenience in writing programs, can make reading difficult. For example, it may not be immediately apparent that "NOT a = b OR c" is equivalent to "(NOT (a = b)) OR (a = c)".

c. It should also be pointed out that the rules given for determining the constituents of an expression are quite awkward; a clean, simple formulation is possible and would be desirable.

d. The changes needed to bring COBOL into compliance with C2 are minor.

3. Expressions Permitted (C3).

a. Degree of compliance: F

b. COBOL fails to meet this requirement. In subscripting, only literals or data-names are permitted; in indexing, only literals, data-names, or data-names offset by a signed literal are allowed [8, pp. I-89-90]. In the PERFORM statement [8, p. II-78], the number of times a procedure is executed, and the lower bounds of the loop control variable, are restricted to identifiers and literals. Similarly, the DISPLAY statement cannot specify expressions [8, p. II-59]. Moreover, there is a somewhat

strange restriction on the form of relation conditions [8, p. II-4.1]: comparison of two literal values is not allowed.

c. Allowing expressions everywhere both constants and references to variables are permitted would complicate the COBOL language definition and would be a large change to existing implementations. COBOL is not oriented towards expressions and allows general expressions only in COMPUTE statements and relations.

4. Constant Expressions (C4).

a. Degree of compliance: F

b. COBOL contains no provision for evaluating constant expressions before run-time.

c. It is not a major change to specify pre-run-time evaluation of constant expressions. Because COBOL uses fixed-point data, no precision errors will result.

5. Consistent Parameter Rules (C5).

a. Degree of compliance: T

b. Since COBOL only has one kind of parameter (effectively, to a procedure), by default the parameter rules are consistent.

c. Although COBOL satisfies this requirement, that is only because the language is so restricted. Were COBOL modified to improve compliance with other Tinman requirements, this requirement would need to be included as a design goal.

6. Type Agreement in Parameters (C6).

a. Degree of compliance: F

b. The simple parameter-passing mechanism in COBOL is essentially "by reference;" no type checking is performed. The only requirement for a match is that the actual and formal parameters occupy an equal number of character positions [8, p. XII-4]. The interpretation of this last requirement, when the formal and/or actual parameter is a "flexible table," is not stated in the defining document.

c. The satisfaction of this requirement is dependent upon revisions to the data type facility of COBOL to bring it into compliance with A1 and A2. Major changes are needed.

7. Formal Parameter Kinds (C7)

a. Degree of compliance: F

b. COBOL allows only one type of parameter, "by reference." Moreover, there are restrictions on the kinds of data which can be passed as actual parameters; according to Syntax Rule (4) [8, p. XII-5], it is illegal to pass a component from a record in WORKING-STORAGE.

c. COBOL has a very limited procedure capability. Design of a more comprehensive facility would encompass the issues in this Tinman requirement as well as those in Tinman requirements C5, C6, C8, C9, P1, P2, G5, G7 and H9. Major changes would be necessary.

8. Formal Parameter Specifications (C8).

a. Degree of compliance: F

b. COBOL does not allow the omission of the specification of the attributes of formal parameters. As stated on [8, p. XII-4] "Each of the operands in the USING phrase of the Procedure Division header must be defined as a data item in the Linkage Section of the program in which this header occurs, and it must have a 01 or 77 level-number."

c. This requirement could best be met with a macro facility of some sort. COBOL has available already the rudiments of such a macro facility in the 'COPY... REPLACING' construct. Were this facility made somewhat more comprehensive (a moderate change to the language), COBOL could be adapted to meet this requirement.

9. Variable Numbers of Parameters (C9).

a. Degree of compliance: F

b. COBOL contains no provision for meeting this requirement. Major changes would be needed to bring COBOL into compliance with C9.

## Section V. VARIABLES, LITERALS AND CONSTANTS

1. Constant Value Identifiers (D1).

a. Degree of compliance: F

b. COBOL contains no provision for associating constant values with identifiers, except for the figurative constants (ZERO, SPACES, etc.) built into the language [8, p. I-81].

c. Such a provision could be added to COBOL without great difficulty.

2. Numeric Literals (D2).

a. Degree of compliance: T

b. COBOL has fairly simple rules concerning literals [8, pp. I-80-81]; the two categories (numeric and non-numeric) cover all the built-in data-types. COBOL complies completely with the D2 requirement.

3. Initial Values of Variables (D3).

a. Degree of compliance: F

b. COBOL partially fulfills this requirement; the VALUE clause [8, pp. II-36ff] is the vehicle for supplying initialization information for data-items. Unfortunately, the rules governing its use are laden with special cases and exceptions: the VALUE clause cannot initialize some kinds of data (such as INDEXES and records containing flexible tables) and cannot be used practically to initialize other kinds. The following illustrates the impractical application just mentioned: since a record is considered alpha-numeric regardless of the PICTUREs of its constituents, the only legitimate initial value which can be supplied in a VALUE clause is a non-numeric literal or a figurative constant. Initialization of tables is similarly restricted; in order to initialize a table of numbers to its constituent values, one must first define a record giving VALUE clauses to its components, and then REDEFINE this with a table -- a somewhat clumsy and circumlocutive arrangement. It is not possible to specify repetition factors (as found, for example, in FORTRAN's DATA statement).

c. COBOL complies with D3's requirement for no default initial values. In the absence of a VALUE clause, the initial value for the item is undefined.

d. COBOL fails to comply with the requirement that "there will be provision (at user option) for run time testing for initialization."

e. As COBOL has already an initializing capability, the major effort needed to conform to this Tinman requirement involves extending the capability to all data types and making the applications consistent.

f. Runtime testing for initialization is very expensive unless supported by hardware.

4. Numeric Range and Step Size (D4).

a. Degree of compliance: P

b. COBOL partially satisifies this requirement via its PICTURE clause [8, pp. II-18 ff]. The reason that COBOL meets D4 only partially is that ranges are restricted solely via the specification of decimal places. Thus data items can be declared to be within range 0 through 999, or within -99.99 through +99.99, but not to be within the range, say, -5 through 12.

c. Bringing COBOL into compliance with D4 would have major impact on both language and implementation. This is partially due to the fact that COBOL deals with data in a variety of formats, and providing range specifiers for each format would be a complex task. COBOL does not have a systematic exception-handling capability; thus, the user would have to define range exception handlers on a statement by statement basis in a similar fashion to the way in which ON SIZE ERROR is treated.

5. Variable Types (D5).

a. Degree of compliance: P

b. There are various restrictions on the way that data can be composed which prevent COBOL from complying with this requirement. For example: tables are restricted to three dimensions; tables must be embedded within records;

condition variables cannot be arrayed; flexible tables cannot be components of other tables.

c. While improvements with regard to this Tinman requirement are possible, full compliance would involve an entire re-orientation of COBOL (as indicated in connection with A1 above).

6. Pointer Variables (D6).

a. Degree of compliance: F

b. COBOL has no pointer facility nor any language-supported means for building data with shared or recursive substructure. In practice, COBOL users deal with some forms of linked data structures externally (e.g., on a disk). The applications for COBOL typically involve quantities of data considerably larger than can fit in memory. Thus, the approach generally taken in COBOL is to process each record individually, keeping only summary information between records.

c. Inclusion of a pointer facility in COBOL would be a substantial extension requiring large design effort. The achievement of data type security would present particular difficulties.

## Section VI. DEFINITION FACILITIES

1. User Definitions Possible (E1).

a. Degree of compliance: F

b. COBOL lacks the facilities needed to allow user definition of new data types and operations. The only definitional features in COBOL are: (1) a facility to define structured data objects (i.e., records and tables), and (2) a limited ability to define callable subroutines. These are in no way sufficient to meet the requirements of E1; implicit in the notion of data type are restrictions on the uses of its member objects, but in COBOL there is complete freedom to use composite data-items simply as alphanumeric values regardless of their structure.

c. The absence of definitional facilities is one of the great weaknesses in COBOL. There is no way to bring the language into compliance with E1 without a nearly complete redesign.

2. Consistent Use of Types (E2).

a. Degree of compliance: F

b. Lacking user-defined types, COBOL fails to meet this requirement.

c. The addition of user-defined types of the kind implied in E1, E5, E6 and E8 is a major change requiring a reorientation of COBOL's data handling facilities. The problem with COBOL here is the language's focusing on the character string as the atomic constituent of (almost) all data structures.

3. No Default Declarations (E3).

a. Degree of compliance: T

b. COBOL satisifies this requirement fully.

4. Can Extend Existing Operations (E4).

a. Degree of compliance: F

b. COBOL has no provision for extending existing operators.

c. Operator extension implies the existence of a type definition facility; the latter is absent from COBOL and would require major changes if it were to be included.

5. Type Definitions (E5).

a. Degree of compliance: F

b. COBOL has no facilities for defining new data types.

c. The provision of a type definition facility for COBOL is a major change and would require redesign of a substantial portion of the language.

6. Data Defining Mechanisms (E6).

a. Degree of compliance: PF

b. COBOL partially fulfills this requirement (with respect to structuring data, however, and not to defining new types). Definition by enumeration of literal names is present to a limited extent in the form of condition variables -- the literal names can only be used in conditions and may refer only to a relation between the enumerated value(s) and one particular data item. (It may be noted that, rather than defining an abstraction independent of its representation, the COBOL construct makes explicit the representation.) COBOL's record and table definition facilities capture, in a restricted fashion, the notion of Cartesian product (but only up to three dimensions for tables). There is no facility for discriminated union in COBOL; nor is there any provision for obtaining power sets.

c. The provision of these facilities in COBOL would be a major change to the language. Again, the problem is COBOL's basic orientation around character data.

7. No Free Union or Subset Types (E7).

a. Degree of compliance: P

b. The various facilities for overlaying data (see A7 above) effectively provide COBOL with free union. There is no type subsetting allowed in COBOL.

c. Restrictions on overlaying data would be a difficult change to make to COBOL, in view of the applications which require much overlaying for efficiency reasons.

8. Type Initialization (E8).

a. Degree of compliance: F

b. COBOL contains no facilities for meeting this requirement.

c. The provision of a type definition facility which satisfies E8 is a major modification to COBOL.

## Section VII. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed (F1)

a. Degree of compliance: FP

b. COBOL is very weak in the area of namescopes; e.g., there is no block-structure, and all data names are known globally throughout the program. The only aspect of COBOL which approaches satisfying the intent of F1 is the "own"-like behavior of data in CALLed programs. As explained in [8, p. XII-5]: "A called program is in its initial state the first time it is called... On all other entries into the called program, the state of the program remains unchanged from its state when last exited. This includes all data fields...."

c. The addition of namescopes to COBOL would be a major change. The problem is that COBOL's PROCEDURE DIVISION is basically linear as opposed to hierarchical in structure.

2. Limiting Access Scope (F2).

a. Degree of compliance: F

b. COBOL contains no facilities for limiting access scope.

c. This Tinman requirement implies a data abstraction capability that does not exist in COBOL. Adding such a capability would amount to a substantial redesign of COBOL.

3. Compile Time Scope Determination (F3).

a. Degree of compliance: FP

b. COBOL partially satisfies this requirement, since most names have their meanings identified at compile-time. The discrepancies between the language and F3 lie in two areas. First, the block structure implied by the requirement that "access scopes will be lexically embedded" is absent from COBOL. Second, some names are only known at run-time. As explained in [8, p. XII-1], COBOL "provides the capability to transfer control to one or more programs whose names are not known at compile time...."

c. Providing block structure in COBOL would be a major change, as mentioned in connection with F1.

4. Libraries Available (F4).

a. Degree of compliance: P

b. COBOL has a limited library feature which serves in effect as a simple macro facility for text insertion. Its main use appears to be a shorthand way of incorporating commonly used file or record descriptions into a program. It is not clear that this satisfies F4's requirement that there be "broad support for libraries common to users of well-recognized application areas."

c. The subroutine capability of COBOL is currently quite weak; thus, there is little advantage to be gained from an object library capability until the language more effectively supports procedures (a major change). On most of the large commercial machines which run COBOL there are general provisions for object time libraries.

5. Library Contents (F5).

a. Degree of compliance: PT

b. COBOL partially satisfies this requirement. The fact that accessing a library entails copying source text implies that the library contents are restricted to COBOL source language; however, the ENTER statement [8, p. II-63] allows code from any implementor-allowed language to be included in the source text.

c. There are no provisions in [8] for compile-time checks at the interface, and, in fact, the facility provided by ENTER is implementation-dependent.

d. The addition to COBOL of security checks on the interface between the language and ENTERed code is relatively minor with respect to language definition; impact on implementation is moderate.

6. Libraries and Compools Indistinguishable (F6).

a. Degree of compliance: T

b. COBOL has no Compool; however, the library facility captures the essential power of the Compool and hence COBOL can be considered to satisfy F6.

7. Standard Library Definitions (F7).

a. Degree of compliance: P

b. COBOL partially satisfies this requirement. It provides interfaces to machine-dependent capabilities (e.g., file descriptions and the ENVIRONMENT SECTION), but does this via a large set of special cases as opposed to a few general rules.

c. The COBOL language could be modified only with difficulty to meet this Tinman requirement. Although COBOL attempts to encapsulate in the ENVIRONMENT DIVISION the distinctions between devices, many differences in file organization and processing must be reflected in the DATA and PROCEDURE DIVISIONs. COBOL tries to achieve program self-documentation through an English-like exposition. However, this has had the effect of emphasizing differences of organization in file and I/O descriptions rather than the features which are common to all I/O.

## Section VIII. CONTROL STRUCTURES

1. Kinds of Control Structures (G1).

a. Degree of compliance: P

b. COBOL partially satisfies this requirement. The sequential control structure is the normal flow of control. For conditional execution, the IF and GO TO ... DEPENDING statements can be used. The PERFORM statement handles iterative execution. There are no facilities for recursion or parallel processing. Several specialized features deal with exception and interrupt handling, primarily in connection with I/O.

c. The addition of recursion and parallel processing capabilities to COBOL would involve major definition and implementation difficulties. The strength of COBOL is that it takes a direct and simple approach to commercial calculations. It is straightforward in COBOL to process single records and to maintain summaries of information from the records processed. COBOL is not directed at real time applications nor at applications requiring sophisticated mathematical calculations; the inclusion of either of those capabilities would require a major effort and imply a different orientation to the language.

2. The GoTo (G2).

a. Degree of compliance: PT

b. The GO TO statement in COBOL satisfies this requirement [8, p. II-65]. It should be noted, however, that in the absence of namescopes COBOL permits GO TO statements to branch to any paragraph (or section) in the PROCEDURE DIVISION. Also, the ALTER statement [8, p. II-57] should not be used; it allows dynamic determination of the target of a GO TO, from a point which may be remote from the actual GO TO statement. This makes the dynamic behavior of a program quite difficult to perceive from its static structure; the GO TO ... DEPENDING statement should be used instead of the ALTER.

c. The ALTER could be easily eliminated as it provides no capability not provided by GOTO ... DEPENDING. The removal of the ALTER statement should simplify COBOL implementations.

3. Conditional Control (G3).

a. Degree of compliance: P

b. COBOL's conditional control structures satisfy this requirement to a high degree. The IF statement [8, p. II-66] is almost "fully partitioned" in the sense of the Tinman; the only exception is that the ELSE clause may be omitted if the IF statement is the last statement in the sentence. Selection may be based on the value of a Boolean expression, or on a computed choice, but not on the subtype of a value from a discriminated union (COBOL lacks discriminated union). A problem with the IF statement is that one cannot transfer control out of a single nested IF statement without transferring out of the entire sentence.

c. The GO TO ... DEPENDING statement [8, p. II-65] serves as a CASE statement. However, COBOL restricts the dispatch expression to be an identifier instead of a general formula. Also, the "fall-through" behavior in the presence of an out-of-bounds identifier is conducive to programmer error.

d. The modifications necessary to bring COBOL into compliance with G3 are not major (unless one interprets this characteristic as requiring a discriminated union facility). Generalizations of the IF and GO TO ... DEPENDING statements are needed.

4. Iterative Control (G4).

a. Degree of compliance: P

b. COBOL's PERFORM statement partially satisfies this requirement. In particular, it allows the repeated execution of one or more paragraphs or sections, either a fixed number of times based on the value of a literal or identifier, or until some condition becomes true. Several of the formats provided for the PERFORM statement are quite readable; e.g., PERFORM ... n TIMES (when a loop control variable is unnecessary), and the AFTER phrase (which nicely reveals the order in which loop variables are varied). However, the UNTIL phrase in the VARYING clause is not as clear as it could be; to execute paragraph P 10 times, with I as loop control variable, one must write: PERFORM P VARYING I FROM 1 BY 1 UNTIL I>10.

c. Some conflicts between COBOL and G4 are as follows. First, COBOL does not permit the loop termination condition to appear at arbitrary points in the loop. In fact, if one wishes to check for a terminal condition, he must code this in the following fashion:
LOOP. ... IF termination-condition THEN GO TO LOOP-EXIT. ...
LOOP-EXIT. EXIT.
and then PERFORM LOOP THRU LOOP-EXIT to effect the iteration. A second conflict is that in COBOL all loop-control variables are global data. Third, it is possible to transfer control into the body of a loop from outside (interestingly, COBOL prohibits the transfer of control in the other direction).

d. In a COBOL program all identifiers are global to the entire program, because there is no way to restrict the scope in the PROCEDURE DIVISION of items declared in the DATA DIVISION. The loop itself is remote from its point of invocation and behaves more as an internally nested procedure than a loop but with the weakness that there is no 'IF done THEN return' possibility. Changing the loop construct would involve large changes to the COBOL PROCEDURE DIVISION.

5. Routines (G5).

a. Degree of compliance: F

b. Recursion is prohibited in COBOL; as stated in [8, p. XII-6], "a called program must not contain a CALL statement that directly or indirectly calls the calling program."

c. COBOL has two mechanisms for dealing with routines. If it is necessary to pass parameters, then the routine must be written as a separate COBOL program, invoked via a CALL ... USING statement. If parameter passing is not needed, then the routine may be written as simply a section or paragraph (called a "procedure" in COBOL) in the program's PROCEDURE DIVISION, invokable via a PERFORM statement.

d. All COBOL storage uses a static or own-like allocation scheme. Modifying the language to use the automatic storage allocation techniques and the displays necessary for recursive procedures would require a major redesign of the language. For typical COBOL applications, recursive procedures offer no particular advantages.

6. Parallel Processing (G6).

a. Degree of compliance: F

b. COBOL contains no parallel processing facilities.

c. Parallel procesing implies that two or more separate procedures, which share at least some data, are simultaneously active. For a language to provide parallel processing facilities means that the shared data must be protected from overlapped update. For COBOL this might mean adding a "SHARED" section to the ENVIRONMENT and DATA DIVISIONS and mechanisms for starting parallel tasks. Such modifications would be a major extension to the language.

7. Exception Handling (G7).

a. Degree of compliance: P

b. COBOL provides a limited set of features for handling exceptional conditions of various kinds. The three basic types of conditions which can be handled are arithmetic overflow, I/O exceptions, and storage bounds overflow. To deal with arithmetic overflow (more precisely, a condition when an arithmetic result exceeds the capacity of the target) the user can supply an ON SIZE ERROR phrase [8, p. II-50] in any arithmetic statement. In the event of overflow, the imperative statement supplied in the ON SIZE ERROR phrase is executed and control then returns to the next statement.

c. A number of features deal with the handling of I/O exceptions. A two-character FILE STATUS data item is available for inspection after most I/O operations to determine the outcome (e.g., Successful Completion, At End, Permanent Error, or Implementor Defined, in the case of Sequential I/O [8, pp. IV-1-2]). The AT END phrase may be supplied to give a programmer-supplied behavior for conditions such as attempting to read when no next record exists [8, p. IV-28]. The USE statement "specifies procedures for input/output error handling that are in addition to the standard procedures provided by the input-output control system" [8, p. IV-28]. A procedure associated with a USE statement is executed after the standard system procedure for dealing with the exception. Procedures associated with USE statements are called

"declaratives" [8, p. I-99] and are logically and physically distinct from "non-declaratives."

d. Another possible exception condition in COBOL is storage overflow during the loading of a CALLed program. "If during the execution of a CALL statement, it is determined that the available portion of object time memory is incapable of accomodating the program specified in the CALL statement and the ON OVERFLOW phrase is specified [in the CALL statement], no action is taken and the imperative-statement [in the ON OVERFLOW phrase] is executed" [8, p. XII-5].

e. Because storage in COBOL is statically allocated, the only excessive memory requirements possible are during a CALL to a non-resident routine, which has an explicit error routine assigned. The arithmetic overflow condition requires the user to specify with each statement the exception handler; thus, there is no transfer of control either forward or backward (unless there is a GOTO). However, this explicit method of handling exceptions offers better opportunities for recovery than, for instance, PL/I, where one process must handle all occurrences of a class of exception conditions.

f. User definition of exception conditions would be difficult in COBOL because the procedure defining mechanism is so weak.

8. Synchronization and Real Time (G8).

a. Degree of compliance: F

b. COBOL was not designed to be a real-time language, and the inclusion of real-time facilities would be a difficult task.

## Section IX. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics (H1).

a. Degree of compliance: P

b. COBOL partially complies with this requirement; areas of conflict are the following. First, COBOL is not free-format; the card-orientation of the language is evidenced by the rules in [8, pp. I-105 ff] (e.g., regarding the indication of continuation lines and the permitted placement of division and section headers). Second, there is an abundance of unique notation for special cases (e.g., the various statements associated with I/O); a simple, uniform grammar appears unachievable. Third, there are actually two types of statement separator: a semi-colon is optional between statements, and a period (followed by a space) is mandatory at the end of a sentence (a sentence comprises one or more statements). Fourth, COBOL allows abbreviations for keywords (e.g., CORR for CORRESPONDING, COMP for COMPUTATIONAL).

c. It might also be pointed out here that the lexical structure of COBOL [8, pp. I-75 ff.] is somewhat more complicated than it need be (especially regarding the placement of spaces). A finer subdivision of the lexical units -- e.g., including non-numeric literals as a token type, in addition to separators and character-strings -- might have been cleaner.

d. Many of the shortcomings that COBOL has in its lexical and syntactic structure are the direct result of the attempt to use English-like sentences wherever possible. Thus COBOL has a large number of noise words, has alternate constructs for singular and plural cases (e.g., RECORD IS and RECORDS ARE), requires a space after periods and provides other examples of clumsiness or verbosity.

e. The changes needed to bring COBOL into compliance with H1 are fairly major in scope.

2. No Syntax Extensions (H2).

a. Degree of compliance: T

b. COBOL fully satisfies this requirement.

3. Source Character Set (H3).

a. Degree of compliance: T

b. COBOL satisfies this requirement. Besides the 26 alphabetic and 10 numeric characters, COBOL includes the following:
space + - * / = $ , ; . " ( ) < >
all of which are included in the 64-character ASCII subset. It should be noted, however, that COBOL allows in comments and non-numeric literals any character in the host computer's collating sequence.

4. Identifiers and Literals (H4).

a. Degree of compliance: P

b. COBOL satisfies this requirement almost completely. The formation rules for identifiers and literals [8, pp. I-75ff] are fairly conventional, except that data names may begin with a numeric character and that the break character for identifiers is the hyphen. (Because of the latter, spaces must appear around arithmetic operators, to distinguish A-B from A - B.) The only discrepancy between COBOL and H4 is that COBOL does not contain a break character for literals.

c. Providing a break character for numeric literals is not a major change; the problem is which character to use. For example, underscore is a possibility, but it can be confused with a hyphen and prints as a left arrow on some terminals.

5. Lexical Units and Lines (H5).

a. Degree of compliance: P

b. COBOL partially complies with this requirement. End-of-line is allowable in non-numeric literals [8, p. I-80]; continuation of lexical units across lines is also permitted [8, p. I-106].

c. The changes needed to have COBOL comply with H5 are minor.

6. Key Words (H6).

a. Degree of compliance: P

b. COBOL partially satisfies this requirement; the primary conflict is that COBOL contains an extremely large number of reserved words (about 300, itemized in [8, pp. I-109-110]). As a result, the programmer must continuously be careful when devising data-names, to avoid conflicts with this set. The opposite problem is also present; the language has in some cases sacrificed clarity by avoiding the use of reserved words. In particular, levels 66, 77, and 88 for data-names have special meanings which are not at all suggested by the numeric values.

c. COBOL programs are intended to be readable by people whose main areas of expertise lie outside the field of computer programming. This goal has led to an inappropriate reliance on English-like constructs (see the discussion of H1 above) and an excessive number of key words, all of which are reserved.

d. It would be very difficult to bring COBOL into compliance with H6 without sacrificing some of the language's basic design goals. The impact on implementation, however, is minor.

7. Comment Conventions (H7).

a. Degree of compliance: P

b. COBOL partially meets the requirements of H7 with the following qualifications. First, there are actually two comment forms in COBOL: the comment-entry used in the IDENTIFICATION DIVISION [8, p. II-2], and the comment line denoted by an asterisk in the continuation indicator area. Second, the contexts in which comments may appear are restricted by the above forms (i.e., one cannot place a comment adjacent to (on the same line as) the statement to which it applies).

c. It would not be difficult to define a comment form for COBOL which does not require an entire line. However, this should be done in conjunction with other redesign which removes the card orientation from the language.

8. Unmatched Parentheses (H8).

a. Degree of compliance: T

b. COBOL complies with this requirement completely. Except when they occur in non-numeric literals and pseudo-text, parentheses "may appear only in balanced pairs of left and right parentheses delimiting subscripts, indices, arithmetic expressions, or conditions" [8, p. I-75].

9. Uniform Referent Notation (H9).

a. Degree of compliance: F

b. COBOL fails to satisfy this requirement. A standard subscript form is present in the language, but a different notation is used for calling routines (programs), viz., CALL program-name USING data-name-1, ... data-name-n. Moreover, there is no facility for calling a routine which returns a value (i.e., a function). In addition, COBOL distinguishes between subscripts and indexes: a subscript is any variable containing an integer, while an index (USAGE IS INDEX) and table (INDEXED BY) must be explicitly declared as such [8, p. I-89]. The difference between subscripts and indices is one of efficiency.

c. Major changes would be required to bring COBOL into compliance with H9; e.g., a "function" subroutine facility would be needed.

10. Consistency of Meaning (H10).

a. Degree of compliance: P

b. COBOL partially satisfies this requirement. One exception is the difference of interpretation applying to adjacent level-01 entries, depending on whether they appear in the FILE SECTION or the WORKING-STORAGE SECTION. (In the former case, storage overlaying is implied; in the latter case, separate storage is allocated.) A second exception is the two uses for the equal sign: equality in relations, and assignment in the COMPUTE statement. As another exception, the hyphen acts as the break character in identifiers, the subtraction operator (a potential source of confusion), and the continuation character for multi-line sentences, entries, phrases or clauses.

c. Different symbols for equality and assignment operators could be used (e.g., '=' and ':='). The hyphen could be reserved for subtraction, underscore used for the break character and one of the 'spare' characters used as a continuation character. The different interpretations of level-01 entries in the FILE SECTION and the WORKING-STORAGE SECTION could be most easily resolved by requiring a REDEFINEs-clause in the FILE SECTION when specifying alternate record formats.

## Section X. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic (I1).

a. Degree of compliance: P

b. COBOL has numerous examples of implementation dependencies and "undefined" situations, in conflict with I1. The action to be taken by a compiler when a syntax error occurs is undefined in [8]. When no SIZE ERROR phrase is specified, the result of an overflow is undefined [8, p. II-50]. When the contents of a referenced data item are not compatible with the class declared for that item in its PICTURE clause, the result of the reference is undefined [8, p. II-52].

d. Full compliance would require that for every undefined action or result the standard specify an explicit action or result. This would be a difficult definitional problem. For some existing compilers the changes required could be very difficult because most of the actions to be defined involve exceptions or errors.

2. Object Representation Specifications Optional (I2).

a. Degree of compliance: P

b. COBOL partially fulfills this requirement, with respect to data representation. The standard (i.e., "default") representation of record components is contiguous character positions irrespective of word boundaries; also, the standard representation of numeric data is DISPLAY (i.e., character code) format. The programmer can override the first default via the SYNCHRONIZED or JUSTIFIED clauses, and the second default via the USAGE IS COMPUTATIONAL clause.

c. COBOL fails to provide any facility for the programmer specifying open vs. closed subroutines. For such a facility to be relevant, a more general subroutine definition mechanism is needed; this is a major change to the language.

d. With no parallel processing or real time capability, COBOL has no need for a reentrant vs. nonreentrant code

generation option. Addition of parallel processing or real time features would be a large extension to the language.

3. Compile Time Variables (I3).

a. Degree of compliance: F

b. COBOL fails to satisfy this requirement. Although the ENVIRONMENT DIVISION includes, in the CONFIGURATION SECTION, paragraphs in which the programmer specifies SOURCE-COMPUTER and OBJECT-COMPUTER, this information cannot be interrogated in the program in the manner required by I3.

c. The figurative constants of COBOL could be extended easily to include configuration information; similarly, some information from the ENVIRONMENT DIVISION could be made available as pseudo-figurative constants. This requirement assumes a compile-time macro capability not available in COBOL (see paragraph I4).

4. Conditional Compilation (I4).

a. Degree of compliance: F

b. COBOL contains no such facilities. The only features which approach conditional compilation are the Debug module (WITH DEBUGGING MODE serves as a compile-time switch [8, p. XI-3]) and the COPY statement with REPLACING clause [8, p. X-2]. The former has the following interpretation [8, p. XI-1]: "When the WITH DEBUGGING MODE clause is specified in a program, all debugging sections and all debugging lines are compiled [normally].... When the WITH DEBUGGING MODE clause is not specified, all debugging lines and all debugging sections are compiled as if comment lines."

c. The COPY statement with REPLACING clause offers a macro-like ability to insert source text in a program while replacing all occurrences of a given character-sequence by another character-sequence.

d. The features described in paragraph b are not sufficient to meet this requirement. They do, however, provide a base that can with moderate effort be extended to provide the conditional compilation facilities required by I4.

5. Simple Base Language (I5).

a. Degree of compliance: F

b. COBOL fails to satisfy the most important aspect of this requirement, viz., the possession of a simple base language. Because of the absence of any extension facilities, the entire COBOL language is in effect the base. The implication of this situation is that powerful and complex features must be built into the language (e.g., SORT, MERGE, STRING, UNSTRING) since there is no way to obtain the desired behavior using the language's definition facilities.

c. In addition, COBOL violates the requirement that each feature provide a "single unique capability". Frequently, COBOL offers several ways of doing the same thing: e.g., in the presence of the COMPUTE statement, the other arithmetic verbs are redundant; synonyms are built into the language (DIVIDE...INTO and DIVIDE...BY, OF and IN, EQUAL and =); optionally specified keywords abound; several different forms are available for declaring the attributes of data-names (the PICTURE clause and USAGE IS INDEX).

d. Although some simplification of COBOL is achievable with only a minor impact on the language (see H1 and H6 above), the ideal of a simple base language is not a realistic goal. A major redesign effort would be required to reshape COBOL to the form envisioned in I5.

6. Translator Restrictions (I6).

a. Degree of compliance: P

b. COBOL partially meets this requirement; specified in the language definition are the following restrictions: the maximum number of table dimensions is 3 [8, p. III-1]; the maximum number of levels in a record is 49 [8, p. I-84]; the maximum number of characters allowed in a PICTURE character-string is 30 [8, p. II-18]; the maximum number of digits in a numeric value is 18 [8, p. I-76]; the maximum size of a non-numeric literal is 120 characters [8, p. I-80]. No restrictions are placed on the number of nested parentheses levels in expressions or the number of identifiers or statements in programs [8, p. I-6].

c. Additional restrictions could easily be imposed on the language and on implementations. The problem is that of deciding what these restrictions should be.

7. Object Machine Restrictions (I7).

a. Degree of compliance: T

b. COBOL fully satisfies this requirement.

## Section XI. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES.

1. Efficient Object Code (J1).

a. Degree of compliance: P

b. COBOL partially meets this requirement. On the positive side, there are several facilities by which the user has direct control over the efficiency of the object code. On the negative side, COBOL has some intrinsic inefficiencies resulting from some of its rules, and it also contains over-general features which have run-time costs even when the full generality is not employed.

c. A variety of features are available to the programmer to control the efficiency of the object program:

(1) The abbreviation of relational conditions almost announces an optimization to the compiler: A < B OR C OR D would obviously be compiled so as to preserve A in a register, whereas some extra effort by the compiler would be required to handle (A<B) OR (A<C) OR (A<D) in the same manner.

(2) The COMPUTE statement can result in more efficient execution than the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements. For example,

```
COMPUTE Z = W * X + Y
```

can be compiled efficiently with no need for a temporary storage location to hold the intermediate result W*X. Using the MULTIPLY and ADD statements, however, would require an additional variable; e.g., MULTIPLY W BY X GIVING TEMP; ADD TEMP, Y GIVING Z.

(3) The use of INDEXes instead of subscripts can improve run-time efficiency.

(4) Specification of COMPUTATIONAL USAGE for data to be used arithmetically will improve run-time performance.

(5) The programmer has direct control of storage overlays via the facilities of the Segmentation module [8, Section IX]; he can also control the

allocation of storage to separately compiled programs via the CANCEL statement [8, p. XII-7].

d. Inefficiency of object code can be caused by COBOL's rules concerning order of evaluation. For example, in the expression (A+B)/C**2, COBOL requires the parenthesized subexpression (A+B) to be evaluated first; the generated code might look like:

```
LOAD A
ADD B
STORE TEMP1
LOAD C
MULT C
STORE TEMP2
LOAD TEMP1
DIV TEMP2
```

If the compiler were free to evaluate C**2 first, there would be no need for the second temporary:

```
LOAD C
MULT C
STORE TEMP1
LOAD A
ADD B
DIV TEMP1
```

e. COBOL also contains facilities whose generality must be "paid for" even when not fully used. Examples are the INSPECT [8, p. II-68ff], STRING [8, p. II-86ff], and UNSTRING [8, pp. II-91ff] statements. The problem is that lacking definition facilities, COBOL must have a large number of features built in which could be obtained by extension in other languages. But to make these features useful to a wide range of programming applications, COBOL defined them in a highly general fashion, with the result that run-time overhead may ensue even when the full generality is not used.

f. Improving the efficiency of COBOL to bring it into compliance with J1 would involve a moderate amount of language redesign.

2. Optimizations Do Not Change Program Effect (J2).

a. Degree of compliance: TU

b. This requirement actually applies to the translator rather than to the language itself; COBOL has no conflicts with J2.

3. Machine Language Insertions (J3).

a. Degree of compliance: P

b. COBOL contains a facility for dropping into machine language (or other high-order languages) -- viz., the implementation-dependent ENTER statement [8, p. II-63]. The restriction that usage of this facility be limited to compile-time conditional statements is not met by COBOL.

c. If the compile time facilities required by I4 were available, this requirement would be satisfied. Inclusion of such facilities in COBOL would be moderately difficult.

4. Object Representation Specifications (J4).

a. Degree of compliance: P

b. COBOL partially satisfies this requirement. Examples of compliance are the SYNCHRONIZED clause (allowing alignment of elementary items on word boundaries, either left or right); the JUSTIFIED clause (permitting programmer override of the normal conventions for positioning whole data-items in receptacles); the USAGE COMPUTATIONAL clause; and the ability to specify "don't care" character positions inside a record as FILLER [8, p. II-15].

c. There are, however, several aspects of COBOL which are in conflict with requirement J3. First, there is no separation of a record's logical description from its object representation; both are defined in terms of character positions within the record. Second, there is no provision for associating a source language identifier with special machine addresses. Third, there are no restrictions on the use of the machine-dependent characteristics of the object representation.

d. The changes needed to meet this requirement would be large, because of the language's orientation around character data.

5. Open and Closed Routine Calls (J5).

a. *Degree of compliance: F*

b. COBOL fails to provide any of the requested features.

c. In the absence of a flexible procedure facility (see paragraph C7 above), this requirement is not relevant. Addition of such a facility would be a large task.

## Section XII. PROGRAM ENVIRONMENT

1. Operating System Not Required (K1).

a. Degree of compliance: F

b. COBOL's I/O facilities, segmentation (i.e., program overlay) feature, and many of its high level capabilities (such as SORT, INSPECT, and COPY) require both a large run-time support system and an operating system.

c. Changes amounting to a redesign of the language would be needed to bring COBOL into compliance with K1.

2. Program Assembly (K2).

a. Degree of compliance: P

b. The COPY feature partially satisfies this requirement in that arbitrary portions of COBOL source text may be inserted in COBOL programs. A weakness of this facility is that the library code, even if fully debugged, can exhibit different behavior in different instances depending on the declared attributes of the data names.

c. Separately compiled programs (in COBOL or other source languages) may be linked to produce a "run unit" [8, p. XIV-40]. However, as noted above in connection with C6 and C7, the procedure capabilities of COBOL are weak, and there is no type checking on parameter passage.

d. A Compool-like library facility (i.e., partial compilation so that attributes are fixed) would solve some of the problems with the COPY feature; this would be a major addition to the language. Inclusion of strict type checking between separately compiled modules would also be a large change.

3. Software Development Tools (K3).

a. Degree of compliance: PU

b. The debug feature of COBOL (see paragraph I4 above) partially satisifies this requirement. The definition of linkers, loaders and other desired tools is outside the scope of [8].

4. Translator Options (K4).

a. Degree of compliance: U

b. The COBOL standard does not address the availability of options such as those required by K4. Their inclusion would not have a large impact on the language but could have major effect on implementations.

5. Assertions and Other Optional Specifications (K5).

a. Degree of compliance: F

b. COBOL offers none of the required facilities.

c. To add such facilities to the language would require major changes to the language design and to existing implementations. As the Tinman notes, there are many opinions as to the desirability, usefulness, and proper form for assertions, assumptions, axiomatic specifications, etc. Until there is widespread agreement on these issues, the inclusion of features such as those listed in K5 in a widely-used ANSI standard language such as COBOL would be ill-advised.

## Section XIII. TRANSLATORS

1. No Superset Implementations (L1).

   a. Degree of compliance: F

   b. Supersets are explicitly allowed in [8, p. I-6]:

      An implementation that includes, in addition to a specified level of each of the functional processing modules and of the Nucleus, elements or functions that either are not defined in the American National Standard COBOL specification or are defined in a given level of a standard module not otherwise included in the implementation, meets the requirements of this standard. This is true even though it may imply the extension of the list of reserved words by the implementor, and prevent proper compilation of some programs that meet the requirements of this standard. The implementor must specify any optional language (language not defined in a specified level but defined elsewhere in the standard) or extensions (language elements or functions not defined in this standard) that are included in the implementation.

   c. Substitution of the prohibition required by the Tinman for the qualification quoted above would be a minor change to the language but could have a large impact on implementations.

2. No Subset Implementations (L2).

   a. Degree of compliance: F

   b. COBOL totally fails this requirement. The language, as specified in [8], comprises twelve modules all of which have a level-1 and a level-2 implementation and some of which have a level-0 (meaning the module need not exist at all). As specified in [8, p. I-4]:

      The full American National Standard COBOL is composed of the highest level of the Nucleus and of each of the functional processing modules. A subset of American National Standard COBOL is any

combination of levels of the Nucleus and of each of the functional processing modules other than the full American National Standard COBOL.

c. Since COBOL was designed to be a subset-oriented language, compliance with C2 would be a major redesign effort.

3. Low-Cost Translation (L3).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard. However, the baroque structure of COBOL and the different syntax conventions used in the different divisions make it unlikely that a COBOL compiler will achieve the goals of requirement L3.

4. Many Object Machines (L4).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in [8]. However, the history of COBOL usage reveals that the language has been implemented for a variety of object machines.

5. Self-Hosting Not Required (L5).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard.

6. Translator Checking Required (L6).

a. Degree of compliance: U

b. The COBOL standard makes no mention of the action to be taken by the compiler in the event of syntax errors or violations of other language restrictions.

7. Diagnostic Messages (L7).

a. Degree of compliance: U

b. The COBOL standard makes no mention of error or diagnostic messages and (as noted under L6 above) there is no requirement that errors be detected. The translator's behavior is implementation-dependent.

8. Translator Internal Structure (L8).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard.

9. Self-Implementable Language (L9).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard.

## Section XIV. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only (M1).

a. Degree of compliance: P

b. As an existing and widely used language, COBOL is composed of features that do not exceed the state of the art. In view of the wide discrepancies between COBOL and the Tinman characteristics, it is not reasonable to expect meeting the following Tinman requirement: "Any design or redesign [to COBOL] which is necessary to achieve the needed characteristics will be conducted as an engineering design effort and not as a research project."

2. Unambiguous Definition (M2).

a. Degree of compliance: P

b. Despite the attempts in [8] to provide a complete and unambiguous specification of COBOL, the complexity of the language makes such a goal virtually impossible to achieve. The following brief set of examples illustrates some of the problems in the document.

(1) The semantics of INDEXes is confusing. If one declares a table to be indexed by a set of data names, the behavior is unspecified when a data name outside this set is used as an index for the table. Also, as noted above in connection with A7, [8] does not explain the effect of REDEFINEing a field which includes an INDEX.

(2) Nowhere in the language definition is there an explicit requirement that all data names must be declared.

(3) "Undefined" situations are frequent (e.g., [8, p. III-6], [8, p. II-50]).

(4) As described in connection with C1 and C2, the language definition confused the independent concepts of the parsing of an expression into constituent subexpressions, and the order in which these subexpressions are evaluated.

(5) As observed in connection with C6, the description of parameter passing is incomplete.

3. Language Documentation Required (M3).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard. However, because COBOL is the most widely-used HOL, introductory and tutorial literature is available. We do not know of any formal definition of COBOL; such a description would be extremely difficult to produce.

4. Control Agent Required (M4).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard.

5. Support Agent Required (M5).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard.

6. Library Standards and Support Required (M6).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard.

## Section XV. CONCLUSIONS REGARDING COBOL

1. Conflicts between Objectives of COBOL and Tinman.

a. Among the six candidate HOLs considered in this report, COBOL is the furthest from satisfying the Tinman requirements. This is not surprising, considering the vast difference between the goals of the two. COBOL was designed as a special-purpose business-oriented language; power was not considered a critical need. The sizable investment in COBOL programs has created a natural conservatism with respect to language modifications. Thus, there is no impetus to make incompatible changes to the language, despite the advances in programming methodology which have revealed better (i.e., more secure and more powerful) constructs than those supplied in COBOL. A result of this inertial effect is that changes to COBOL have tended to be of the "add-on" variety, and interactions with other features have led to a large and complex language.

b. The Tinman's "needed characteristics," on the other hand, are oriented to general purpose programming and attempt to incorporate the current state of the art in the production of reliable software. Obviously, there is no commitment in the Tinman to the constraint of compatibility with existing languages.

2. Summary of Major Areas of Conflict between COBOL and Tinman.

a. Data and Types. The absence of a Boolean data and the fact that the existence and treatment of floating point are implementation-dependent are serious deficiencies in COBOL. The clumsiness of the array facility is also a major drawback. Even in COBOL's selected area of business data processing, the lack of any secure method of processing records with alternate structures is a serious handicap.

b. Operations. The presence of implicit conversions and the absence of explicit conversions, and the lack of operations for dealing with power sets or bit strings, resulted in serious conflicts with the Tinman. COBOL also offers little in the area of scalar operations.

c. Expressions and Parameters. The main failure is with respect to C6 (Type Agreement in Parameters); COBOL lacks

type checking on parameter passage. Also, COBOL does not generally allow expressions in contexts in which constants and variable references appear (C3, Expressions Permitted), nor is there a facility for pre-run-time evaluation of constant expressions (C4, Constant Expressions). The discrepancies with respect to C8 (Formal Parameter Specifications) and C9 (Variable Numbers of Parameters) are less critical.

d. Variables, Literals, and Constants. COBOL does not allow the association of constant values with user-supplied identifiers, as required in D1 (Constant Value Identifiers). Apparently, such a facility did appear in previous versions of the language (the CONSTANT SECTION of the DATA DIVISION) but is not present in X3.23-1974. Also, COBOL lacks a pointer facility (D6, Pointer Variables).

e. Definition Facilities. The shortcomings of COBOL are most apparent in this area; COBOL does not permit the definition of new data types. Thus the language fails to comply with E1 (User Definitions Possible), E2 (Consistent Use of Types), E5 (Type Definitions), or E8 (Type Initialization). It also lacks the ability to extend existing operators (E4).

f. Scopes and Libraries. The absence of block structure is a serious deficiency in COBOL, especially with respect to the construction of large programs. This absence is reflected in COBOL's failure to meet requirement F2 (Limiting Access Scope).

g. Control Structures. COBOL's deficiencies in the area of real-time applications are revealed here; the language lacks facilities for parallel processing (G6) and for synchronization (G8). Also, there is no support in COBOL for recursive routines (G5).

h. Syntax and Comment Conventions. COBOL lacks a notation for uniform reference (H9). COBOL is not free format and has many special forms.

i. Defaults, Conditional Compilation and Language Restrictions. COBOL's deficiencies in this area are a lack of compile-time variables (I3) and conditional compilation (I4), and its failure to have a simple base language (I5).

j. Efficient Object Representations and Machine Dependencies. COBOL's deficiency here is relatively minor; it lacks a facility whereby the user can distinguish between open and closed subroutine calls (J5).

k. Program Environment. The major conflict with the Tinman in this area is COBOL's requirement for an operating system and large run-time support (violating K1). Also, COBOL lacks a variety of facilities, in connection with software development tools, translator options, and assertion-like specifications, which are required by the Tinman.

l. Translators. Major conflicts here arise because COBOL is designed to have subsets and supersets (in violation of L1 and L2), the complexity of the language makes low-cost translation an unrealistic goal (in violation of L3), and the reference document does not require translator checking or standardize a set of error messages (violating L6 and L7).

m. Language Definition, Standards, and Control. These requirements do not actually pertain to the language; we note, however, that the COBOL defining document has serious conflicts with M2 ("Unambiguous Definition").

3. Unnecessary Features in COBOL.

a. There are a large number of COBOL features unnecessary for meeting Tinman requirements and many other features which are realized in the language via redundant forms. Examples of the former are the INSPECT statement, the Sort-Merge Module, editing PICTUREs, the SEARCH statement, and the ALTER statement. Of these, we specifically recommend deletion of the ALTER statement in view of the difficulties this statement raises with respect to program verification and maintenance, and because the main effect of the ALTER statement can be achieved via the GO TO ... DEPENDING construct. As for the other features named, they are present in COBOL primarily because of the weakness of the language's definitional facilities. If they were to be eliminated, a considerable sacrifice in notational convenience would have to be made. As a result, we do not recommend their deletion. In view of the fact that some of these facilities are "special purpose" in nature, we recommend that COBOL's subset-oriented approach be preserved, despite the Tinman's opposition (L2).

b. The redundant forms in COBOL are illustrated by separate constructs which provide the same effect (e.g., the arithmetic and COMPUTE statements, or the MOVE statement and some options of the arithmetic statements), by "noise" words, and by abbreviations and spelling variations for the same keyword. Although the attempt here was to promote readability, the opposite effect can unfortunately result, since the person who maintains a program must be familiar with all the variations employed. We recommend the removal of redundant forms, and suggest that, until this is done via the language definition, COBOL project managers select a single variation for each form to be employed by the programmers.

4. Problems Concerning Verification of COBOL Programs.

A recent study [10] sponsored by the Army's Computer Systems Command treated the feasibility of formally proving the correctness of COBOL programs. Aspects of the COBOL language which were specifically cited for their difficulty with respect to verification are the ALTER statement [10, p. 13]; the Debug, Sort-Merge, and Communication Modules [10, p. 14]; storage overlaying [10, p. 18]; truncation [10, p. 26]; consideration of data items as strings; and implementation-defined language features [10, p. 28]. A conclusion of [10] was that the major problems encountered concerning COBOL verification were "verbosity of the programs, assertions, and verification conditions" and "the semantic complexity of the COBOL language". This conclusion is consistent with our evaluation of COBOL in this chapter.

5. Recommendations concerning COBOL.

On the basis of the evaluation conducted in this chapter, we conclude that an attempt to modify COBOL to bring it into compliance with the Tinman would be inadvisable. The many fundamental differences between the two, caused by the basically different objectives desired, would imply a substantial redesign effort and a new implementation. It is not realistic to expect the "flavor" of COBOL to be retained in such a new language. Thus, the advantages typically cited for choosing an existing language (availability of implementations, documentation, trained programmers) would effectively be forfeited. In summary, COBOL in its present form is not suitable with respect to satisfying the Tinman requirements, and no language which differs from COBOL in only minor ways will be substantially better.

# CHAPTER 8

# PL/I EVALUATION

## Section I. LANGUAGE SUMMARY

### 1. Lexical Properties.

PL/I is a free-format language based upon the EBCDIC character set. A fifty-six character subset is used for token formation [2, Section 2.5]; however, in special contexts such as character literals and comments any character defined in the implementation may be used. The space character separates tokens. PL/I's 188 keywords are not reserved. Identifiers may be of arbitrary length, but an implementation may restrict the maximum length provided that the maximum is at least 31 characters. The underscore may be used as a break character in identifiers. PL/I comments are delimited by "/*" and "*/", and any implementation-defined character may appear in a comment. Comments cannot be nested; so, although code may be included in a comment, the included code may contain no comments.

### 2. Data Types.

The data types available in PL/I can be roughly categorized as shown in Figure 7.

a. Scalars. The two scalar categories are computational and non-computational. PL/I offers type-checking only between the computational and non-computaational types and among the non-computational types. For example, it is illegal to assign a label value to a floating point variable or a pointer variable.

(1) Behavioral properties of all non-computational data types.

(a) Assignment is defined for POINTER and OFFSET, ENTRY, LABEL, FORMAT, AREA, and FILE variables. On assignment PL/I will convert between POINTER and OFFSET, but in all other assignments involving non-computational types, the source and the target have the same type. (In effect, POINTER and OFFSET may be considered the same type.)

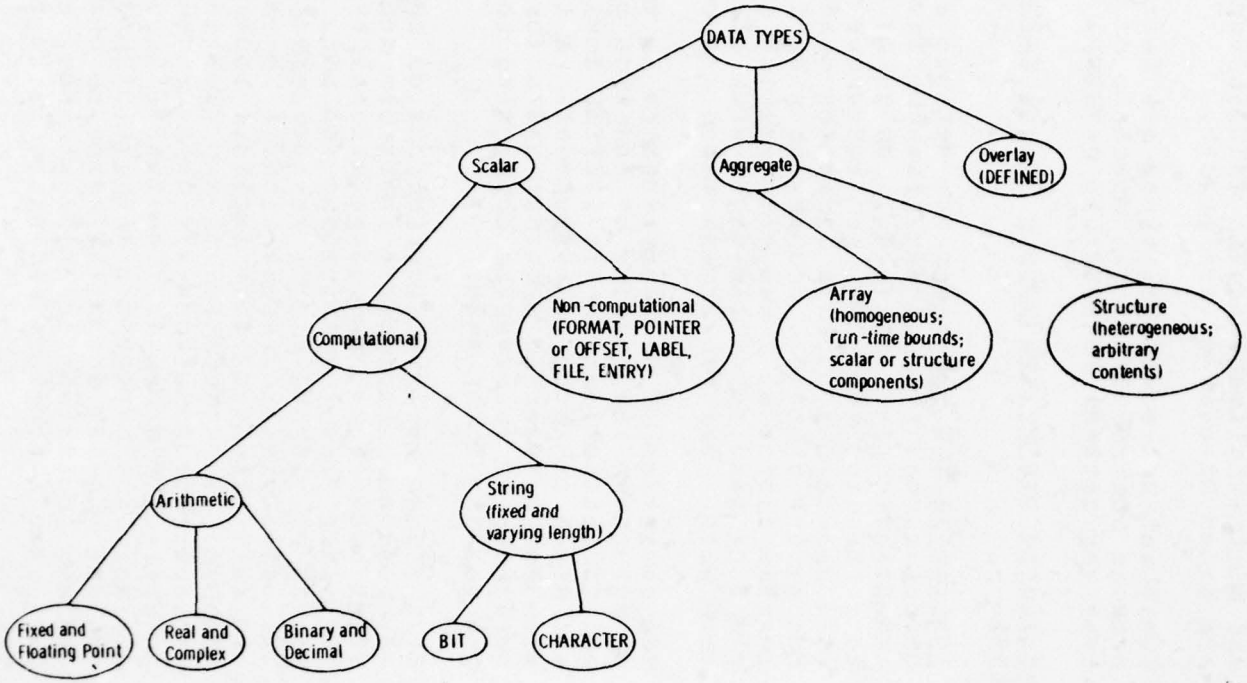



Figure 7. Data Types in PL/I

(b) The semantics of value parameters to procedures, of returning the results of function procedures, and of initializations (in INITIAL declarations) are based on that of assignment.

(c) Equal and not-equal relations are defined between non-computational operands of the same type (or between POINTER and OFFSET).

(2) Behavioral properties common to all computational types.

(a) Assignment and relations are defined among all computational types. Any mismatch involving an undefined conversion (e.g., assigning '12A.34' to a floating variable) or high-order truncation will generate a runtime exception condition. For string operands, assignment when fixed lengths differ will cause truncation on the right or padding on the right with blanks or zero bits. The conversion rules are complex.

(b) The semantics of value parameters to procedures, of returning the results of function procedures, and of initialization (in INITIAL declarations), are based on those of assignment. A type mismatch between an argument and a formal parameter is resolved by converting the argument and passing it by value.

(c) PL/I places no type restrictions on operands in expressions. Implicitly-invokable conversions exist between any two computational types. These are "true" conversions and not, as in JOVIAL or TACPOL, simply re-interpretations of the underlying representations. We emphasize the full extent of this implicit conversion process: concatentation of two floating variables causes those variables to be converted to character strings. The result is a character string. The expression (INT||FLOAT)**INT will convert the INT and FLOAT types to character strings, concatentate those strings, convert that result to float and raise it to the power INT.

(d) There are a number of built-in conversion routines, which give the PL/I user the ability to explicitly control precision, length and other attributes of the result.

b. Aggregate Data Types. PL/I has extremely flexible facilities for defining homogeneous or heterogenous data structures. Arrays may have any number of dimensions, and components may be of any scalar data type or any structure. Structures may have any scalar, array or structure components.

(1) Properties common to all aggregate data type.

(a) Component Selection occurs via indexing (for arrays) and via dot-qualified name (for structures). Only sufficient qualification to assure uniqueness is required.

(b) Compatible aggregate data types may appear in relational, string or arithmetic expressions; the operations are performed on a component-by-component basis. There are some restrictions: neither operand may contain a component of a non-computational data type and the two aggregate types must be compatible. See comments under paragraph J1 for more description. Note that "compatible" does not mean the two aggregate data types "look alike" (e.g., any computational scalar is compatible with all structures and arrays having no components of non-computational type).

(c) Assignment is defined for aggregate data types if the source is promotable to the target.

(d) Alignment is a representational property of aggregate types: the user may declare an aggregate UNALIGNED to minimize storage required or ALIGNED to minimize execution time.

(2) Arrays. PL/I permits the declaration of arrays of any rank of any component type (including structure). Bounds may be specified either at compile-time or at run-time. An array slice is permitted wherever an array of the slice's rank is permitted.

(3) Structures. A structure is PL/I's version of what the Tinman calls a record. PL/I permits the declaration of structures with components of any scalar type (computational or non-computational), array type or structure type. Each component has a name. Component selection is through dot-qualified and indexed identifiers (e.g., A(3).B.C(5).D, which could also be written A(3,5).B.C.D or A.B.C.D(3,5) as well as in other ways). Qualification sufficient to make the name unique is required; i.e., in the example above if D appears only once in the entire namescope, then D(3,5) selects the same component as A.B.C.D(3,5).

c. Untyped Data. PL/I contains several facilities which defeat the type checking for scalar data.

(1) The UNSPEC may appear as the target of an assignment.

(2) The DEFINED declaration (PL/I's version of EQUIVALENCE or OVERLAY) has the effect of a free union; it permits instances of scalar or aggregate types to be overlayed on the same storage area.

3. Procedures.

a. Function Procedures and Procedures. PL/I allows the definition of procedures. Procedures that have a RETURNS attribute are "function procedures" and may be used everywhere a reference is required. Procedures having no RETURNS attribute may be used only in CALL statements.

b. Formal Parameters. A procedure or function procedure may have any number of parameters. Each parameter may be of any scalar (computational or non-computational) or aggregate type. Function procedures may return values of any scalar or aggregate type.

c. Arguments. Non-computational arguments must match the type of the corresponding parameter (correspondence is by position in the argument or parameter list). The types of computational scalars or aggregates need not match. If they do not match, implicit conversions are performed as discussed under assignment. Converted arguments are passed by value; if the called procedure returns some result in one

of its parameters, then a type mismatch will cause these results to be lost with no exception condition raised. The result of an expression in the argument list is clearly passed by value; through the use of parentheses the user converts any reference into an expression and thus forces the argument to be passed by value. When attributes of parameters and arguments match exactly, the argument is passed by reference.

4. Exception Condition Handling.

a. PL/I provides a mechanism allowing the user to obtain control upon the occurrence of any of a set of language-defined, implementation-defined or user-defined exception conditions. The implementation-defined conditions relate to certain I/O and operating system actions that are dependent on configuration and hardware. The user gets control of an exception by executing, before occurrence of the exception, an ON-statment (e.g., ON FIXEDOVERFLOW: {<single statement>|<begin block>}). If, and whenever, a FIXED OVERFLOW occurs, then the <statement> or <begin block> is executed; it is executed only for exceptions and not executed sequentially. Note that control is transferred backwards, logically if not in the actual source text, when an exception occurs. After execution of the on-unit, control returns to a point just before or just after the code causing the exception unless there is an explicit transfer of control (a GOTO) from the ON-unit, in which case the ON-unit and any active blocks embedded in the block containing the ON-unit are terminated. Any condition may be raised through use of the "SIGNAL condition name;" statement. The user-defined conditions may be raised only through this means.

b. Computational Conditions are raised in the following:

(1) CONVERSION: The source for a conversion contains a character which cannot be converted to the target type (e.g., '12A.34' cannot be converted to any numeric type). The user can use ONSOURCE and ONCHAR to read and change the offending source string or character.

(2) OVERFLOW: Floating-point exponent exceeds implementation maximum.

(3) STRINGRANGE: Substring referenced with SUBSTR does not lie within the specified string.

(4) SUBSCRIPTRANGE: A subscript expression is not within declared array bounds.

(5) ZERODIVIDE: Fixed or float divide-by-zero error.

(6) FIXEDOVERFLOW: Truncation of high order bits or digits.

(7) SIZE: On formatted output or assignment, high-order digits are being lost.

(8) STRINGSIZE: On formatted output or assignment, a string value is too long to be represented in the specified field or variable.

(9) UNDERFLOW: A floating point computation yields a value not equal to zero but which is too small to be represented in the object machine.

c. I/O Conditions. In PL/I an I/O condition is always associated with a particular named file. I/O conditions are raised in the following circumstances:

(1) ENDFILE: A READ or GET statement has read the end-of-file marker.

(2) ENDPAGE: For a PRINT file only, the user specified number of lines per page has been output.

(3) NAME: On executing a GET DATA statement the name in the input streams did not match any expected name.

(4) KEY: During a KEYed access operation, a key error has occurred.

(5) TRANSMIT: On any input or output operation there is a physical error (i.e., hardware error) which prevents completion of the requested transfer.

(6) UNDEFINEDFILE: During an open operation the requested file was not found or could not be created.

(7) RECORD: The program has attempted to transfer a record of a size unacceptable to the implementation or of a different size from the destination variable.

d. General Exception Conditions. These are conditions which can be raised for errors in processing that are neither computational nor I/O. These conditions are also raised on the occurrence of an exception condition for which the user has provided no explicit condition handler. General exception conditions are raised in the following circumstances:

(1) ERROR: Raised for any exception condition for which there is no explicit condition handler. ERROR is also raised when an explicit condition handler does not correctly process a condition.

(2) STORAGE: Raised when there is not adequate free storage available during a procedure prologue or an allocate statment.

(3) FINISH: Raised immediately prior to normal or abnormal program termination. In the absence of a FINISH condition handler, the program is terminated.

(4) AREA: Raised when an attempt to allocate a BASED data object within an AREA variable fails because there is insufficient remaining storage in the AREA.

e. Programmer-Defined Conditions. The user may declare a condition name, may use that condition name in an ON statement, and SIGNAL occurrence of the condition at any point in the program.

5. Statements.

a. Null Statement. This statement permits extra semicolons to appear in programs.

b. Assign Statement. The semantics of assignment was described above; the PL/I assignment statement permits multiple target variables which are assigned to in left-to-right order (e.g., I=4; A(I),I,A(I) = 5; sets A(4), I, and A(5) to 5).

c. Call Statement. This statement is used to invoke procedures which have no RETURNS attribute.

d. GOTO Statement. The PL/I GOTO statement may be used to transfer control to any label whose name is known in the scope of the statement, or, through a label parameter or label variable, to any label in an active block. Transfers out of procedures are possible but jumps into nested blocks are not. Arrays of label constants are possible and may be used for an effect similar to the FORTRAN computed GOTO.

e. IF Statement. The PL/I IF-statement may have either one alternative (following THEN) or two alternatives (one following THEN and the other following ELSE). The expression following the IF is eventually evaluated to a bit-string and a high-order 1-bit is the "true" condition.

f. DO Statement. The PL/I DO statement may serve two different purposes. It may be used simply as parentheses for a group of statements which is to be treated as a single statement but without the overhead of entering a nested begin block (e.g., a group of statements to be executed for the THEN clause of an IF-statement). The other purpose is to allow iterative execution of a sequence of statements. It can do this in a variety of ways: by executing while some Boolean expression is true; by stepping a loop control variable from an initial value by fixed amounts until a terminal value is reached; by evaluating an expression after every iteration and assigning the result to a loop variable or by a combination of the three. The loop control variable may be of any computational type.

g. Begin Blocks. A sequence of declarations and statements may be bracketed by BEGIN and END to form a begin block, which may then be used anywhere a single statement is valid. A begin block delimits a namescope.

6. Storage Allocation.

PL/I has four storage classes, each of which implies a different allocation strategy. STATIC storage is allocated before execution. AUTOMATIC storage is allocated as part of the prologue to a begin block or procedure. AUTOMATIC storage is de-allocated, FREEd, on normal or abnormal exit from a begin block or procedure. BASED and CONTROLLED storage are allocated by an explicit programmed ALLOCATE

statement and de-allocated by a FREE statement. Objects of all four storage classes are initialized at allocation time if the INITIAL attribute is coded as part of the declaration. STATIC strings and arrays must have their bounds fixed at compile-time, but strings and arrays of the other three storage classes may defer resolution of their bounds until allocation time.

7. Files and I/O.

a. One of the most attractive features of PL/I is the variety of features available for processing external data. The two major classifications of data files are STREAM and RECORD. STREAM files are character oriented; RECORD files are record (or block) oriented.

b. STREAM files are input, output or printed. Printing assigns some additional attributes such as page size, i.e., maximum number of lines on the page. STREAM files are processed using GET (read) or PUT (write) statements. The user may specify any of three different modes: EDIT, which performs formatted I/O; LIST, which performs I/O using default format specifiers; and DATA, which requires or supplies variable identifiers to perform I/O using default format specifiers.

c. RECORD files are collections of records, each of which contains a single scalar or aggregate data object stored in the machine's internal representation. There is no conversion during RECORD I/O. Record files also have one of the attributes DIRECT (i.e., random) or SEQUENTIAL, and may also have a KEYED attribute, which specifies that records are to have an associated key allowing the program to specify the next record to be processed.

d. In addition to the GET, PUT, READ, WRITE, and REWRITE (i.e., update) statements outlined above, PL/I has OPEN and CLOSE, DELETE (which deletes a record) and LOCATE (which allows the user to build an output record directly in the output buffer).

e. No data type information is preserved with a file, a fact which makes it possible to WRITE data of one type and READ it back as another. A file may be OPENed for input with attributes entirely different from those used in the creation of the file. Misuse of this behavior may cause an error at some point during file processing.

f. A number of exception conditions for I/O are defined. See paragraph I.4.c above.

8. Compile-Time Facilities.

a. PL/I has a %INCLUDE statement [2, Section 2.5.7] but no action for this statement is defined in [2]. In existing implementations, %INCLUDE inserts source text from a named file into the program; typically, this source text includes system-wide declarations and procedure definitions.

b. PL/I as defined by [2] provides no facilities for macros, conditional compilation, or COMPOOLS.

c. Although [2] defines the translation and processing with no reference to separate compilation, existing implementations may and do allow for separate compilation.

## Section II. DATA AND TYPES

1. Typed Language (A1).

a. Degree of compliance: P

b. PL/I partially satisfies this requirement in that an associated set of "attributes" fully specifies each PL/I variable and each component of a composite data structure; additionally, the attributes specify the precision, the storage allocation mechanism to be used (see P1 below), namescope, and, to a certain extent, the internal representation of a variable or component of a composite data structure. The attributes (and thus the type) of each variable are fully determinable at compile-time.

c. PL/I fails this Tinman requirement by:

(1) Allowing default declarations of attributes. Indeed, the attribute structure of PL/I is so complex that explicitly declaring all the attributes would be an onerous burden on the user.

(2) Providing implicit type conversions, which effectively remove the possibility of using type information for redundancy.

(3) Providing free union (the DEFINE) facility. There is no provision at run time for determining which alternative of a union is the active one. Certain uses of based storage effectively provide free union.

d. The following modifications would improve the conformance of PL/I to the Tinman requirement.

(1) Require explicit declaration of all type attributes (e.g., FLOAT, FIXED, BINARY, DECIMAL, PRECISION, etc.). The factoring allowed in the PL/I DECLARE statement removes part of the burden this modification might have for the user.

(2) Fix in the standard some storage class and namescope assumptions (e.g., AUTOMATIC, INTERNAL), which would then require explicit overrides. This approach also saves keywords.

(3) Remove all implicit conversion from the language. All conversions must then be explicitly specified using the built-in functions.

(4) Apply restrictions to the DEFINE facility (see A7 below).

e. The major impact upon implementation of these modifications is to simplify the construction of PL/I compilers, which would be somewhat smaller, faster, and more reliable. These modifications should also improve the quality of programs written in PL/I. Aside from d(4) above, these restrictions do not constrain the user in the operations that can be performed; only in how those operations are presented and how the work is divided between the compiler and the user.

f. It may be noted that the restrictions and modifications described above are contrary to some of the basic design goals of PL/I.

2. Data Types (A2).

a. Degree of compliance: PT

b. PL/I offers integers and floating values with implementation defined maximum precision, and fixed point real numbers with implementation defined maximum, precision and scale. PL/I provides bit and character strings which may be of fixed or variable length; bit and character strings have an implementation defined maximum length, and the variable length strings have a user-defined maximum length. PL/I offers the user the ability to define arrays and records (called STRUCTURES in PL/I), as required by the Tinman.

c. PL/I does not offer an explicit BOOLEAN type but uses a bit string of length 1 (see B6 for implications). Provision of an explicit Boolean type in PL/I would be straightforward.

3. Precision (A3).

a. Degree of compliance: P

b. PL/I partially satisfies this requirement. A default precision may be specified to apply to all floating point variables in a namescope [2, Section 4.3.6]; such specification is not required. PL/I does not require precision specification for floating point variables; if not specified, precision defaults to some implementation-defined value [2, Section 4.3.6.3]. Precision applies to operations: an operand of an operator is converted to the precision of the operand with greatest precision [2, Section 9.3.2.1ff].

c. This Tinman requirement could be met by requiring that the default precision for a namescope be specified. A single specification at the outermost level would then suffice for an entire compilation. This is a relatively minor change.

4. Fixed Point Numbers (A4).

a. Degree of compliance: P

b. The fixed point numbers offered by PL/I have a range and a fractional step size which are specified by the user (or defaulted, as described above in connection with A1). This range is expressed as the number of bits or decimal digits used to represent the number. Thus FIXED DECIMAL (4,2) allows all values in the range -99.99 to +99.99. There is no way to restrict further the range (e.g., to between -50.00 and +50.00). The scale factors are managed by the compiler.

c. PL/I does not provide support for range restriction and testing. It is difficult to see how such a feature could be added to PL/I without great expense.

5. Character Data (A5).

a. Degree of compliance: P

b. PL/I offers no general facility for defining enumeration types. It is not possible, for instance, to use characters, as "characters," for index values. However, some of the effects of treating character sets as enumeration types are achieved: PL/I defines all the relational operators on single characters and on character strings, and PL/I does offer a number of built-in functions

allowing the programmer great flexibility in the processing of character data, including the following: translating between internal code and any specified code or between any two specified codes; finding the position of one string in another string [2, Section 9.4.4.35]; determining the collating sequence on the host machine [2, Section 9.4.4.14]; and selecting any specified substring of a given string [2, Section 9.4.4.69].

c. The modifications needed so that enumeration types can be defined are described in connection with paragraph E6 below. It should be pointed out, though, that an attempt to obtain the character data type by such a facility is not likely to be cleanly achieved. One problem is that a single character currently is obtainable as a character string of length 1, but there is no corresponding concept of a string of objects from an enumeration type. Associated with this problem is the issue of how a character string literal can be defined, since in general there is no literal form for arrays of enumeration types.

6. Arrays (A6).

a. Degree of compliance: P

b. PL/I requires explicit specification of the number of dimensions, and of the range of subscript values (unless the lower bound equals 1). The type of array component may be specified through default values [2, Section 4.3.6.3]. The number of dimensions and the type are determined at compile-time. PL/I allows the programmer to defer determination of both the upper and the lower subscript bound [2, Section 3.1.1 (A18, A129) and Section 7.2.7] until entry to array allocation scope. Only a contiguous subsequence of integers is allowed as subscript values [2, Section 7.2.7, Step 1.1]. An unspecified lower bound defaults to one -- not zero [2, Section 4.4.3.3, Step 2].

c. The modifications to PL/I required to meet this requirement either depend on other features (e.g., definition of an enumeration type) or are straightforward (e.g., fixing the lower subscript bound at compile-time or changing the default lower bound from one to zero).

7. Records (A7).

a. Degree of Compliance: F

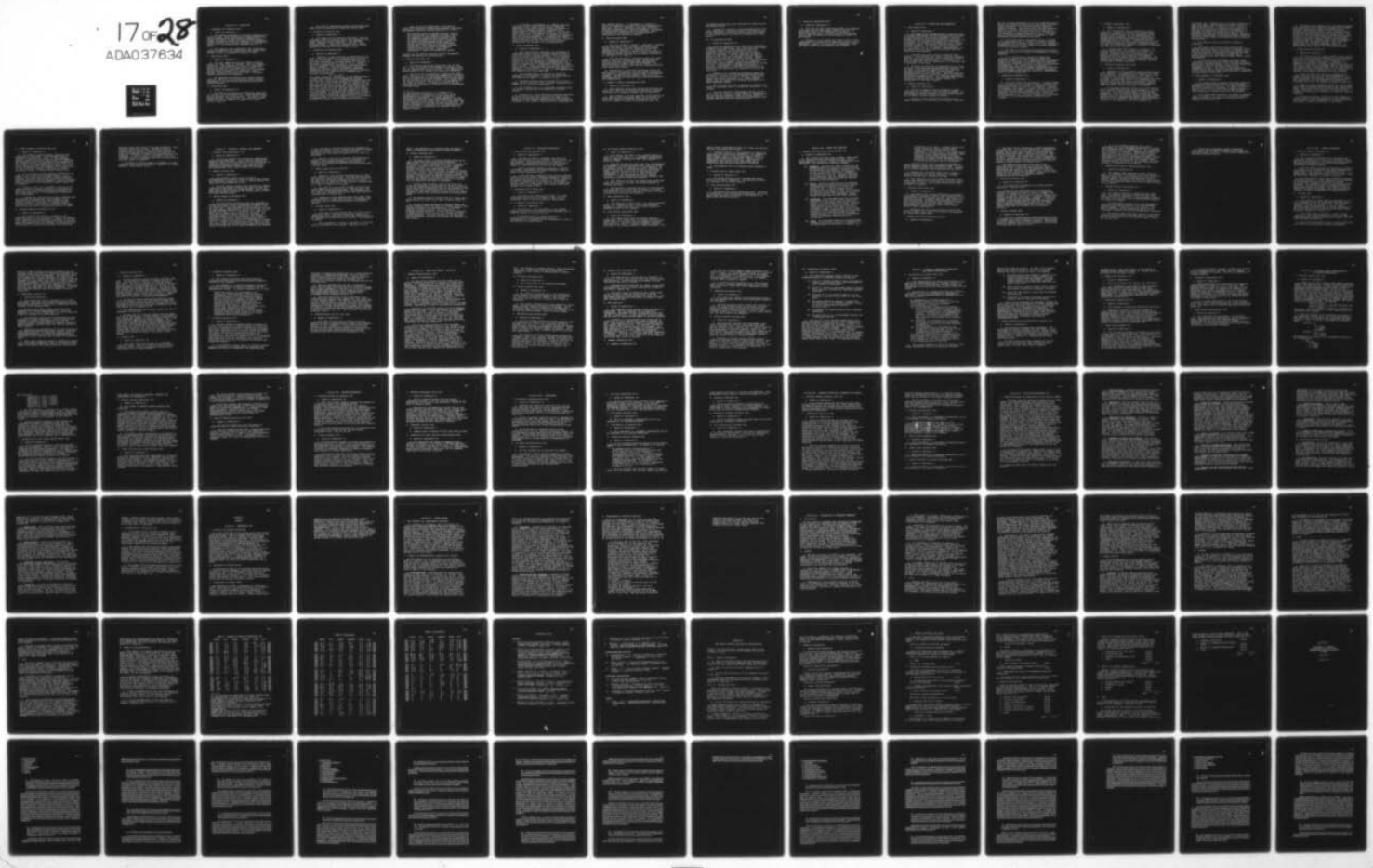
b. Although PL/I permits records to have alternate structures, it fails to satisfy the Tinman requirement because no discrimination condition is required; thus the PL/I DEFINED attribute merely allows the same storage to be interpeted differently using different names. For instance, a PL/I programmer may declare a STRUCTURE, and DEFINE an alternate form and then use components from both the base STRUCTURE and the DEFINED alternate in the same expression. Type checking is defeated; the entire mechanism has very low security.

c. To fully satisfy this Tinman requirement, PL/I needs a modified declaration form to specify the alternate STRUCTUREs and the discrimination condition, and both compile-time and run-time enforcement mechanisms. These are major changes.

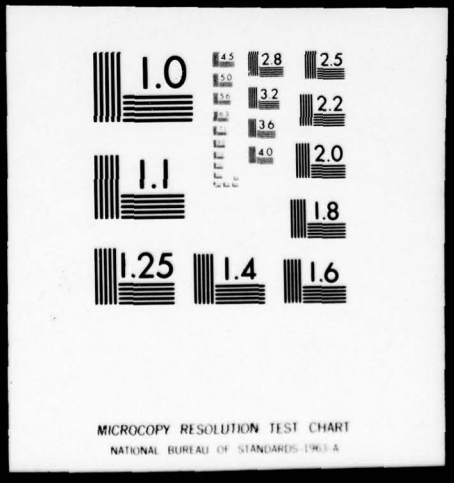
AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

17 OF 28
ADA037634



17 OF 28
ADA037634



## Section III. OPERATIONS

1. Assignment and Reference (B1).

a. Degree of compliance: P

b. PL/I defines assignment and reference operations for all data types. These operations are extended beyond the Tinman requirements in that automatic conversion (e.g., integer to float or character to numeric) and reformatting (termed "promote"ing in the PL/I standard) of structures [2, Section 7.5], are automatically performed (see also J1 below).

c. PL/I does not offer encapsulated type definitions; allowing encapsulated type definitions would require major changes to PL/I, as described in Section VI below.

2. Equivalence (B2).

a. Degree of compliance: P

b. PL/I will compare any two data objects (records, arrays, or atomic data) for identity. PL/I will convert -- implicitly -- all components of records or arrays before making the comparison [2, Sections 6.3.5.1 and 9.1.5]. Considering only the atomic data types, PL/I will compare any pair among floating point, fixed point, integer, picture, character string, and bit string -- performing whatever conversions are necessary.

c. The comparison of floating point values requires identity. The set of built-in functions could be easily supplemented to provide comparison functions having the required behavior.

3. Relationals (B3).

a. Degree of compliance: PT

b. PL/I defines the standard six relational operations [2, Section 4.4.4] for numeric data. These operators apply also to character and bit string data. PL/I does not have enumerated types. Unordered data (e.g., pointer or label values) may only be compared for equality [2, Section 9.3.2.5.1 Case 3].

c. The scope of modifications needed to add enumeration types to PL/I is described in connection with E6 below.

4. Arithmetic Operations (B4).

a. Degree of compliance: T

b. PL/I offers all the requested operations. When the programmer wishes to explictly control the precision of the result there are built-in functions -- ADD, MULTIPLY, DIVIDE, SUBTRACT -- which allow him to do this. When a programmer wishes to obtain the remainder from an integer division the MOD built-in function must be used.

5. Truncation and Rounding (B5).

a. Degree of compliance: F

b. Truncation is also considered in connection with B9 below. Range and precision are attributes of variables in PL/I, not of programs. In an assignment operation, the right-hand expression is evaluated before the attributes of the target are determined [2, Section 7.5.1]. Further, within an expression the evaluation is eventually between pairs of operands. In an infix-expression involving operands of different types a conversion is performed considering only the types of the two operands [2, Sections 9.1.5 and 9.3.2.1]. Thus, if A is a floating point variable, the statement

```
A = 13 + 1/4;
```

will assign 3.25E0 to A because the 1/4 is evaluated first to fixed point 0.2500...0 to implementation maximum precision with a scale factor of maximum precision minus one (i.e., (N, N-1) in PL/I-like notation) [2, Section 9.5.1.9, Step 3.2, and Section 9.3.2.4]. Then the addition will be performed and the result (precision, scale factor) are such that there is only one digit position before the decimal point [2, Section 9.3.2.1]. The most significant position is truncated and the "fixed-overflow-condition" is raised, if it is enabled. An argument could be made that the above operations are not within the range specifications of the program; however, fixed point division in PL/I is inherently so confusing that is is not possible to intuitively determine the result; hence the operation is error-prone.

c. There are several difficulties to be faced in changing the behavior of fixed-point arithmetic operations. One problem is defining suitable behavior [15, p. 87]:

> The precision rules are probably the part of PL/I that has received the greatest criticism. Much of the complaint has been supported by results that are apparently anomalous when considered outside the design philosophy of the rules. A great deal of effort has been expended in searching for a simpler and improved set of rules, particularly for division. Many different proposals have been examined, each has its own set of anomalous results and none has shown a sufficient improvement to justify an incompatible change.

Probably any well-defined behavior could be implemented; defining that behavior is the difficulty.

6. Boolean Operations (B6).

a. Degree of compliance: P

b. PL/I Boolean operations include only AND, OR, and NOT. A user can define the NOR operation in terms of these but this operation is not part of PL/I. A NOR operation on Boolean values or bit strings could easily be implemented.

c. AND and OR are not evaluated in short-circuit mode (*) [2, Section 6.3.5.1] because any expression is evaluated before (or independent of) any determination that the result will be assigned to a Boolean variable. Also, Boolean does not exist as a distinct type in PL/I but is represented by bit string of length 1. In longer bit strings only the high-order bit is examined [2, Section 6.3.5.1] to determine a Boolean result.

---

(*) The following is stated in [2, Section 1.2.1, Flexibilities of Interpretation 1.2.1.3.(4)]: "If an implementation can determine the result produced by evaluating an <expression> ... without evaluating it, the implementation need not perform the evaluation." Although this allows short-circuit mode, there is no requirement that Boolean expressions be evaluated in short-circuit mode. For reasons discussed in connection with B5, determination that short-circuit mode is appropriate is difficult under the language standard.

d. If Boolean is implemented as a separate data type then short-circuit mode could be easily implemented. Operations on bit strings cannot be short circuited; it would involve compiler overhead to determine that fixed-length bit strings were of length 1. For variable-length strings run time checking is required to find if short circuit mode is possible. To achieve full compliance with this Tinman requirement, the standard must require short-circuit mode rather than just allowing it. Without an explicit Boolean data type this would be a difficult modification; with Boolean, it is straightforward.

7. Scalar Operations (B7).

a. Degree of compliance: P

b. PL/I permits scalar operations and assignments on conformable arrays and data transfers between records or arrays of identical logical structure. PL/I also offers more: one-for-one compatibility of type is not required; records are not required to have identical form (see [2, Section 9.1.1.4] for definition of "compatibility" and [2, Section 7.5.3.1] for definition of "promotability") but rather be "compatible" or "promotable" depending on usage (see J1 below). Arrays involved in operations must have the same rank and each dimension must have the same number of components [2, Section 9.1.1.6, Step 2]. The components are not required to be of the same type.

c. PL/I multiplication of matrices is "unnatural, confusing and inconvenient" in exactly the manner used by the Tinman as an example [2, Section 9.1.1.6].

d. Transfers between arrays or records may be used to permit run time conversion from one object representation to another (e.g., to pack or unpack data).

e. The BY NAME option on an assignment statement allows the complete reformatting and conversion of data in a record.

f. Although PL/I allows obscure and confusing operations on arrays and records, great care must be taken in the design of restrictions (or redesign of the base features) because the highly complex behavior shown in the example in paragraph J1, for instance, results from the compounding of

much simpler behavior. If assignment of a scalar to a record is disallowed then much of the error-proneness also goes away. This is a sufficient restriction and would simplify implementation of new PL/I compilers. The changes to existing PL/I compilers would also be fairly minor, involving only restrictions in the definitions of compatibility and promotability applied.

8. Type Conversion (B8).

a. Degree of compliance: P

b. PL/I will attempt to convert -- implicitly -- between any two data types in the following list: float, fixed, integer, bit string, character string and pictured data (special form of character) [2, Sections 9.5ff]. The conversion from character to numeric will give an error if the character string contains any non-numeric character(s) [2, Section 9.5.1.7, Case 3].

c. PL/I provides explicit conversion operations: among integer, fixed and floating point data (FIXED and FLOAT built-in functions) and between object and character representations of numbers (GET and PUT STRING). The built-in function FIXED is also used for conversion between fixed point scale factors.

d. Restrictions on mixed mode expressions violate the spirit of PL/I but would not, in fact, reduce the power of the language or involve much extra writing in well-designed programs. The modifications needed to bring PL/I into compliance with B8 are relatively minor and should simplify the implementation.

9. Changes in Numeric Representation (B9).

a. Degree of compliance: PT

b. PL/I restricts ranges only through the precision and scale factor declared for fixed-point variables. Explicit conversion is not required between ranges so restricted.

c. PL/I provides a run-time exception (fixed overflow or size) when the most significant digits of any integer or fixed-point value are truncated [2, Sections 9.1.4 and 9.1.4.1]. However, as noted in paragraph B5 above, the

circumstances under which such truncation can occur are not intuitively obvious.

d. Although PL/I provides the desired behavior for this Tinman requirement, restricted ranges in the sense desired are not available; the addition of such range restrictions to the language would be a difficult modification to make cleanly.

10. I/O Operations (B10).

a. Degree of compliance: PT

b. One of the strong points of PL/I is the flexibility provided for I/O operations. PL/I files have either STREAM or RECORD attributes. Stream operations imply the processing (either reading or writing) of the next sequential character or compound operations processing groups of characters -- the data processed may only be character data. The PL/I programmer may treat character strings and stream files in a similar fashion. Record operations imply the processing of data in fixed or variable sized units (i.e., records). Records are transferred to/from the external medium without modification [2, Section 8.6.1.2]. PL/I provides an implementation-defined attribute, ENVIRONMENT, to describe any peculiarities of devices or file organization [2, Sections 1.2.1.6.(1) and 1.2.2.(5)].

c. Whether a PL/I program may dynamically assign and reassign I/O devices is an implementation-defined feature [2, Section 1.2.2(23) and (27)]. However, a file may be opened with certain attributes (e.g., OUTPUT STREAM), closed, and then reopened with different attributes (e.g., INPUT SEQUENTIAL RECORD).

d. The PL/I user may elect to process any specific I/O exception conditions and/or to process all exceptions in his program.

e. The only significant disagreement between B10 and PL/I is that PL/I allows a certain degree of implementation and installation dependence, which the Tinman prohibits. Complete implementation and installation independence would present some very difficult design problems.

11. Power Set Operations (B11).

a. Degree of compliance: P

b. PL/I does not offer enumeration types, so it does not offer power sets of those types. However, the PL/I bit string can be used by the programmer to represent user-defined power sets. The PL/I built-in function BOOL allows a user to define any bitwise operation between two bit strings.

c. Because bit string operations already exist in PL/I, implementation of power sets of enumeration types is greatly simplified. However, the definition of enumeration types and their behavior in PL/I would be quite complex.

## Section IV. EXPRESSIONS AND PARAMETERS

1. Side Effects (C1).

a. Degree of compliance: P

b. The Tinman allows implementations to alter the order of argument evaluations when the alteration does not change the side effects of left-to-right evaluation. The PL/I standard explicitly allows certain potential side effects to be ignored. One potential side effect of an expression evaluation is the processing of exceptions (e.g., overflow or divide by zero). The standard does not require these exception conditions to come in any order [2, Section 1.2.1.4, items 1, 2, and 4]; moreover, implementations are not required to raise the exception conditions which are called for in the standard.

c. Functions are another source of potential side effects. If the function F(Y) alters X then the expression X**2+F(Y) could have different values depending on which of X**2 and F(Y) were evaluated first. [2, Section 1.2.1.4 item 3] states that the sub-expressions of an expression may be evaluated in any order.

d. The order of evaluation of expressions in PL/I could be fixed. That would make optimization more difficult. If the order of side effects (except those that arise from exception conditions) is fixed, then many useful optimizations still remain available. This would make new implementations easier to build and would not be difficult for existing implementations. Even compilers which perform substantial optimizations will usually disable the optimizations at the user's request.

2. Operand Structure (C2).

a. Degree of compliance: P

b. The PL/I arithmetic operator hierarchy is widely recognized and is similar to that provided by FORTRAN. The non-arithmetic operator herarchy is not so widely familiar but is similar to that provided by FORTRAN.

c. Because a string concatenation operation is not widely implemented, its position in the operator hierarchy

may not be widely recognized, but its PL/I assigned position between the arithmetic operators and the Boolean operators seems appropriate. PL/I has eight distinct levels of operator hierarchy. Operations within the same precedence level (e.g., X/Y/Z) will be parsed left-to-right (e.g., (X/Y)/Z) except at the highest level (e.g., X**Y**Z) where the operations are parsed right-to-left (e.g., X**(Y**Z)). This is standard FORTRAN and widely recognized; it also represents standard mathematical practice.

d. PL/I allows explicit parentheses either to specify the intended parsing or for clarity. PL/I never requires parentheses in an expression: regardless of how confusing an expression may be to a human reader, it is never ambiguous to the PL/I standard compiler.

e. PL/I offers no facilities for defining new operator precedence rules nor for changing the precedence of existing operators.

f. If the restrictions on mixed mode expressions discussed in connection with B8 were in effect, the potential complexity of a PL/I expression would be lessened. Another great improvement in clarity could be achieved by requiring the different classes of operators to be delimited by parentheses. For example, when the operands of a relational expression are arithmetic expressions, they must be parenthesized irrespective of the relative precedence between arithmetic and relational operators. These changes could be easily implemented.

3. Expressions Permitted (C3).

a. Degree of compliance: T

b. PL/I fully meets this requirement. A careful reading of [2, Section 2.4 and 2, Chapter 3] finds no instance of allowing references to variables of a particular type where expressions of that type are not also allowed. Constants are required, for example, to specify the precision of fixed point and floating point variables, the picture of picture variables, and the replication factor for a replicated string constant (e.g., the string 'AAAAAA' may be specified as (6)'A'), and in these positions variables are not allowed.

4. Constant Expressions (C4).

a. Degree of compliance: P

b. The PL/I standard does not address this point explicitly. However, [2, Section 1.2] ("Relationships Between an Implementation and This Definition") taken together with [2, Section 1.2.1.4] ("Flexibilities of Interpretations Expression Evaluation") implies that an implementation has the option of evaluating constant expressions prior to run time. Although the PL/I standard could specify evaluation of constant expressions prior to run time, the constraints imposed in paragraph D2 might make such evaluation very difficult.

c. In places where PL/I requires a constant in the concrete syntax (e.g., [2, Section 2.4.4.1 CM27]), a constant expression would be a syntactic error. PL/I requires constants in few places; allowing constant expressions in those places would not be difficult. The problem is that of deciding on which operations should be compile-time evaluable in constant expressions (division? exponentiation? library routines?).

5. Consistent Parameter Rules (C5).

a. Degree of compliance: P

b. Parameters to PL/I procedures may be passed by value or by reference. All constants [2, Section 4.4.5.9 Create Entry Reference], expression results, substructured arrays [2, Section 4.4.5.10 Test Matching, Step 1], and all variables or aggregates requiring type conversion are passed by value. When the types (i.e., PL/I attributes) match, variables and arrays are passed by reference. When the forms of records and the types of all the corresponding components match, records are passed by reference.

c. An exception condition process (i.e., PL/I ON-unit) may be a single statement (not an if-statement [2, Section 2.3.3 CH9]) or a begin block, neither of which accepts parameters. There are built-in functions (ONCODE, ONCHAR, ONFIELD) which are only defined during an exception process. ONCODE returns an implementation defined value indicating the cause of the exception. ONCHAR and ONSOURCE change the contents of the character or the field causing the

conversion error. ONFIELD returns the entire contents of an input field when a data-directed I/O statement finds an unknown variable label [2, Section 8.7.1.5]. The use of these built-in functions as (essentially) arguments to on-units, and particularly their use as the target of an assignment statement, are inconsistent with the rules applicable for parameter passing to procedures. The non-I/O exception conditions have no parameters (except ONCODE) and can have no direct effect on the operation which generated the exception.

d. PL/I does not offer any parallel processing facilities.

e. The built-in functions of PL/I obey the same parameter passing rules as user-defined procedures, except that the MAX or MIN function can accept an argument list of arbitrary length and certain built-in functions may appear on the left side of an assignment statement (PL/I pseudovariables: IMAG, REAL, STRING, SUBSTR, UNSPEC, ONCHAR and ONSOURCE).

f. To bring PL/I into conformance with this Tinman requirement, the parameters to exception handling routines must be more carefully defined. To do so would involve some difficulty in both the definition phase and the implementation phase. The nature of the definitional problem can be seen in trying to determine where a parameter to an exception routine should come from, what it should do, and what should be done with it.

6. Type Agreement in Parameters (C6).

a. Degree of compliance: P

b. PL/I matches actual arguments and formal parameters by performing any conversions on the actual arguments necessary [2, Section 6.3.6.1.2]. The number of dimensions for array parameters is determined at compile time [2, Section 3.5.5]. In PL/I the size and subscript range for array parameters are passed as part of the arguments.

c. PL/I allows type transfers to be hidden in procedure calls [2, Section 6.3.6.1.2]. PL/I allows the subscript ranges for arrays to be implicit on the call side [2, Section 6.3.6.1.2 Step 2.3].

d. The rules against implicit type conversions discussed under paragraph B8 are sufficient to assure type agreement of actual arguments and formal parameters when all procedures are compiled together. The PL/I standard [2] assumes that all procedures are compiled together and does not discuss separate compilation and linking steps. Any changes to this larger environment could be very expensive. A linker in a large system is generally shared by many different compilers -- FORTRAN, COBOL, PL/I and Assembler -- so not only must the changes be made to the linker but possibly also to the other languages. Thus, the definitional phase presents some difficulties but the implementation could involve very large system changes.

7. Formal Parameter Kinds (C7).

a. Degree of compliance: F

b. PL/I offers the first formal parameter class (call by value) but does not disallow assignment to these parameters. Any change in the procedure to the parameter will have no effect on the argument, which may be a constant, an expression or a variable enclosed in parentheses. PL/I offers the second class of parameter (call by reference) but care must be taken by the programmer that the attributes of the argument and the parameter agree, or PL/I will perform a conversion and pass the converted value in a temporary location, thus creating the effect of a call by value.

c. It may be noted that control of argument class resides with the caller and not with the procedure. The caller can ensure call by value, which protects a variable from modification, but the procedure has no method of specifying that data to be modified is passed by reference.

d. There are formal parameter classes for labels, which may be used as exception condition exits, and for procedure names. User defined exception condition names remain valid within a namescope; thus, control in PL/I need not pass through the procedure call interface to process an exception condition.

e. Substantial changes to PL/I and to its supporting routines (e.g., linker) are required to allow procedures to specify which of their parameters will be modified and for the system to assure no violations to this intent.

8. Formal Parameter Specifications (C8).

a. Degree of compliance: P

b. PL/I offers an explicit generic facility in the definition of a procedure. Such a procedure must be internal [2, Section 4.3.6.1 Step 4]. The programmer defines a number of alternate entry points to a procedure; the translator selects the appropriate entry depending on the attributes of the arguments and uses that entry as the entry point to the named procedure [2, Section 4.4.5.11]. If the attributes of the call arguments do not match any of the sets of generic parameter attributes the compiler recognizes an error [2, Section 4.4.5.11 Step 1.2.5].

c. It is not clear that the capability required by the Tinman in fact offers the facilities desired. It is, however, quite clear that PL/I offers no "attribute-free" parameter class which would allow a procedure to build a stack or queue with arbitrary contents. A generic procedure with entry points for all conceivable argument types would be unwieldy.

d. It may be noted that a programmer could use PL/I's generic facility to suspend conversion, by specifying one entry with the calling argument attributes desired. Any deviations would result in an (implementation-dependent) error announcement.

e. A compile-time text substitution macro facility would best satisfy this requirement. Some versions of PL/I provide a macro facility, which was excluded from the standard because it is difficult to rigorously define. Provision of such a macro facility would not change other parts of PL/I but would be moderately difficult in itself.

9. Variable Numbers of Parameters (C9).

a. Degree of compliance: F

b. PL/I requires the number of arguments in the subroutine call or function reference to equal the number of formal parameters of the procedure [2, Section 4.4.5.9 Case 2.1]. There are some built-in functions having an arbitrary number of arguments (e.g., MIN and MAX) but this facility is not available for user-defined routines. PL/I does allow

different named entry points to the same procedures. When a different entry point is used, a different number of arguments may be supplied. With the generic feature, a single procedure name may be used with the selection of entry point dependent on number and characteristics of the arguments. In neither case is it possible in PL/I for a programmer to write a procedure which may be called with an arbitrary number of arguments.

c. Allowing an arbitrary number of arguments to a PL/I procedure, even with the restrictions imposed by the Tinman, requires a large change to PL/I.

## Section V. VARIABLES, LITERALS AND CONSTANTS

1. Constant Value Identifiers (D1).

a. Degree of compliance: F

b. PL/I has no facility for representing computational constant value identifiers. Such a facility seems to be within the spirit of PL/I and should not be too difficult to implement, especially considering that PL/I does have a facility for processing constant value identifiers for non-computational values such as labels or file names.

2. Numeric Literals (D2).

a. Degree of compliance: P

b. PL/I provides a syntax and a consistent interpretation for literals of built-in data types; however, the relationship between the values of program literals and input data is not considered in [2].

c. The PL/I language standard could specify that numeric literals and input data of exactly the same form have the same value on any particular target machine. Such a change could have an effect throughout the entire compiler if host and object machines differ.

3. Initial Values of Variables (D3).

a. Degree of compliance: P

b. The PL/I initial attribute allows the programmer to specify the initial values of individual variables, arrays or records as part of their declaration [2, Section 3.1.5 A24, A25, A26]. These variables are initialized at the time of their apparent allocation [2, Section 5.3.6 Step 1.1.2.3; Section 6.2.3 Step 2.1.1.3; and Section 7.2.1 Step 1.3]. There are no default initial values for variables. For variables without the initial attribute, the standard generates variables with an "undefined" initial value [2, Section 7.2.9 Step 1.2]. These variables have certain contexts in which they "must not" be "undefined" [2, Section 7.5.2.1]. However, the interpretation of a "must" or "must not" test is that any failure of these tests means that the program interpretation is no longer governed by the standard

-- i.e., any action, including ignoring the condition, is allowed [2, Section 1.2(1)]. Although the standard provides no explicit provision for testing for variable initialization, such testing is implicit in the "must not" condition.

c. Testing at run-time for variable initialization is extremely costly unless supported in the hardware. To attempt static compile-time checking would result in compilers of much greater complexity and still achieve only a partial check.

4. Numeric Range and Step Size (D4).

a. Degree of compliance: P

b. Inherent in the precision specification of a fixed point variable is the step size. PL/I ON-conditions allow the testing of subscript ranges [2, Section 7.6.7 Step 4.2] but this does not imply any restriction on the range of a variable; only a restriction on the range of a variable when it is used as subscript to an array.

c. The range of a fixed point variable in PL/I is the entire range of its precision (e.g., with precision (7,3) the range is -9999.999 to +9999.999). The hardware limits of a particular implementation are the only range limits on floating point or integer data.

d. Inclusion of range specifications and runtime range checking would be a large change to PL/I, interacting with facilities such as fixed point arithmetic and parameter passing.

5. Variable Types (D5).

a. Degree of compliance: P

b. PL/I does not allow defined types nor does it allow enumeration types. With these exceptions, there are no restrictions on the structure of data; PL/I allows arrays to be components of records and allows records to be components of arrays.

c. This requirement is wholly in the spirit of PL/I and, were the unavailable types present, compliance would be

total. Type definition is discussed under paragraph E1 below and enumeration types under paragraph E6 below.

6. Pointer Variables (D6).

a. Degree of compliance: P

b. PL/I does provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. A PL/I pointer may point to any form of array or record or to scalar variables of any type, including pointer [2, Section 3.1.19, A170 and A172]. Pointer variables may only point syntactically to variables, records, or arrays explicitly declared as based [2, Section 4.4.5 Case 3.1.(2)]. Any other security checks on pointers and based variables require expensive runtime overhead. For example, the data object dynamically referenced by a pointer variable is required to have certain attributes in common with the data object syntactically pointed to [2, Section 7.6.3]; testing for this match is expensive.

c. Because based variables, records and arrays are explicitly allocated by the program, the scope of allocation is not statically determinable; they may be allocated at any point within their declaration scope [2, Section 7.2.3]. A weakness is that based variables, records, or arrays remain allocated outside their declaration scope unless explicitly freed.

d. The standard does not specify what is to occur when a based variable is "lost" (i.e., when no pointer points to it).

e. The PL/I pointer mechanism was designed before the current knowledge about the impact of language features upon software reliability had been developed. It provides capability without constraint and is basically a hardware-oriented mechanism. It would be almost impossible to impose security on the current PL/I pointer and based storage mechanism. Any major redesign of this mechanism would have effects throughout the language and compilers for the language.

## Section VI. DEFINITION FACILITIES

1. User Definitions Possible (E1).

a. Degree of compliance: P

b. PL/I does not offer a programmer the ability to define new data types or new operators, infix or prefix, in his programs. Data definition in PL/I is limited to specifying the structure of objects; the power and added safety of a type definition facility are not present.

c. The PL/I procedure defining capability is flexible and general, particularly the generic facility, which is useful in many application areas.

d. A type definition capability using solely procedures, dynamic and static storage allocation schemes with built-in data types and structuring is already available. Extension of existing operators to new types (e.g., a type which invokes a procedure when a data object of the defined type is the target of an assignment) would require substantial changes to the design of PL/I.

2. Consistent Use of Types (E2).

a. Degree of compliance: F

b. PL/I has no type definition facility. If a type definition facility were to be added to PL/I then this requirement could be included among the design goals.

3. No Default Declarations (E3).

a. Degree of compliance: F

b. The type (i.e., PL/I attributes) of any program component not explicitly declared by the programmer will be given by default [2, Section 4.3.6.3].

c. As discussed above in connection with A1, restrictions on default definitions should be straightforward to implement but are antithetical to some of the original PL/I design goals.

4. Can Extend Existing Operators (E4).

a. Degree of compliance: F

b. PL/I defines the effect of the existing operators with all valid data types (e.g., arithmetic values may be added or compared; pointers or labels only compared) [2, Section 9.3.2ff].

c. Although not under the user's control, many operators are extended to include the record type. When two records of dissimilar form are operands of an infix operator, an intermediate record is constructed using a complex set of rules. If this can be done the two records are called "compatible" [2, Section 9.1.1.4]. Both records are "promotable" to the intermediate record [2, Section 7.5.3.1]. All infix operations may involve automatic type conversions, again not under user control.

d. PL/I offers no facility for defining new atomic data types and no facility for user controlled extension of existing operators.

e. The extension of existing operators to newly defined data types must be included with the design of an entire type definition task, and provision of an operator extension facility makes this even more difficult.

5. Type Definitions (E5).

a. Degree of compliance: F

b. PL/I currently provides none of the features required in E5. The inclusion in PL/I of data type definition features is a large task that would involve redesign of substantial parts of the language.

6. Data Defining Mechanisms (E6).

a. Degree of compliance: P

b. PL/I offers no facility for defining types by enumeration. PL/I offers arrays of records and records of arrays nested to any depth, and the components may be any PL/I type (e.g., fixed or floating arithmetic variables, character or bit strings, pointer or label variables). PL/I

offers neither discriminated union nor a power set facility, but PL/I does provide bit strings.

c. Different methods for the definition of enumeration types have been proposed and implemented. The simplest of these methods, using a unique literal name to specify a small integer constant in restricted contexts, could be implemented without great difficulty. Power sets of enumeration types may be easily implemented when an enumeration type definition mechanism is available.

d. Discriminated union is discussed under paragraph A7 above.

7. No Free Union or Subset Types (E7).

a. Degree of compliance: P

b. The DEFINE facility of PL/I provides free union. PL/I does not support subsetting. See paragraph A7 above for discussion of the DEFINE facility.

8. Type Initialization (E8).

a. Degree of compliance: F

b. The user of PL/I cannot define new types. Inclusion of initialization and finalization procedures as part of the type definition should be included in the design of the entire type definition facility.

## Section VII. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed (F1)

a. Degree of compliance: PT

b. PL/I follows the scope rules of ALGOL. Thus a name may be referenced within its declaration block and all blocks nested in that block in which the declaration is not overridden. PL/I offers four allocation schemes:

(1) Automatic. On entry to a block, storage for the program structure is allocated and any requested initialization is performed [2, Section 6.2.3 Case 2.1.1]; on exit from the block all automatic storage allocated for that block is deallocated [2, Section 6.2.4 Step 2]. Automatic variables may be referenced from the allocation block and internally nested blocks in which the variable name has not been redeclared.

(2) Static. Storage is allocated and any requested initialization performed as part of the program initialization [2, Section 5.3.2 Step 4 and Section 5.3.6]. A static variable may be referenced from the block in which it is declared and from all blocks nested in that block in which the variable has not been redeclared. Static storage is equivalent to ALGOL "own" variables.

(3) Controlled. The programmer explicitly allocates controlled storage in the block in which it is declared or in any block nested in that block [2, Section 7.2.2 and Section 3.1.14 A92]. Only the latest allocation of controlled storage may be referenced [2, Section 7.6.1 Case 2.4]; thus, controlled storage is effectively a stack. When the block in which controlled storage is declared terminates, any controlled storage allocated remains allocated [2, Section 6.2.4] and defined and may be referenced when that block is reactivated.

(4) Based. The programmer explicitly allocates based storage in the block in which it is declared or in any block nested in that block [2, Section 7.2.2

and Section 3.1.14 A92]. A pointer together with storage identifier selects the particular allocation of based storage which is to be referenced [2, Section 7.6.2]. All allocations of based storage are simultaneously available. When the block in which based storage is declared terminates any based storage allocated remains allocated [2, Section 6.2.4] and defined and may be referenced when that block is reactivated.

c. Although static, based or controlled storage can remain allocated outside their declaraion scope, they can not be accessed outside their declaration scope. (However, use of pointer variables allows exceptions to this rule.)

d. Because PL/I follows the scope rules of ALGOL, a nested block wishing to restrict access to a declared program structure must re-declare the name.

e. Full compliance with this Tinman requirement would require the restrictions on pointers that are also required by paragraph D6 above. These restrictions could be quite difficult to implement.

2. Limiting Access Scope (F2).

a. Degree of compliance: P

b. The names of data program components are accessible as described in paragraph F1 above. Data declared static or controlled may also be declared external, which gives it an unrestricted scope. Procedure names are known only in the block in which they are defined, unless declared external, which gives them an unrestricted scope. However, redeclaration of access scope via PL/I block structure partially meets requirement F2.

c. Design of the entire type definition facility (see Section VI above) must include the design of the mechanism for restricting access scope.

3. Compile Time Scope Determination (F3).

a. Degree of compliance: P

b. The scope of PL/I identifiers is wholly determinable at compile-time. PL/I does not require that identifiers be declared but when they are declared they may be declared anywhere within their scope. Keywords and variable names may be the same with only a contextual distinction. (E.g., 'DO DO=1;' is legal. The second 'DO' is the loop control variable.) Access scopes in PL/I are lexically embedded with the most local definition applying when the same identifier appears at several levels [2, Section 4.4.5].

c. The modifications needed to bring PL/I into compliance with F3 include the restrictions on default declarations (see A1 above) and the requirement that all declarations must appear at the beginning of a block (i.e., before the first executable statement). These are not major changes to the language, but modifying existing implementations may be difficult.

4. Libraries Available (F4).

a. Degree of compliance: PU

b. PL/I offers many mathematical and string handling functions [2, Section 9.4.4].

c. Because of the method used in [2] to define the language, the creation and use of object or compile-time libraries is outside the scope of the PL/I standard. For example, [2, Section 1.4.3.2 Step 2.1] states: "Obtain, from a source outside this definition, a sequence of characters composing a putative PL/I external procedure ..." The PL/I standard, which attempts absolute rigor and unambiguity, could be usefully supplemented by some sort of a "PL/I System Implementers Guide" describing in greater detail features which must be available, as, for instance, libraries.

5. Library Contents (F5).

a. Degree of compliance: PU

b. PL/I has an implementation-defined %INCLUDE facility [2, Section 1.2.2 (8) and Section 2.5.7] which could be used to provide compile-time accessible libraries. This feature cannot be evaluated for this study because the standard does not define what its effect is to be.

c. As in the preceding paragraph (F4), PL/I's conformance with this requirement could better be assessed with a different style of documentation. It is certainly the intention of the standards committee that compile-time source libraries be available. Compile-time libraries containing PL/I source code and object-time libraries containing both PL/I object procedures and object routines whose bodies are written in other source languages are available in existing commercial PL/I systems. Finding a suitably rigorous definition is the only difficulty with inclusion of libraries in the standard.

6. Libraries and Compools Indistinguishable (F6).

a. Degree of compliance: FU

b. The comments under paragraph F5 above apply here also. The inclusion of uncompiled source code, provided by %INCLUDE, offers much lower security than a partially compiled COMPOOL, because the included source text can acquire different attributes from different contexts while the partially compiled COMPOOL will have the same attributes in all contexts.

7. Standard Library Definitions (F7).

a. Degree of compliance: P

b. PL/I offers a general I/O capability with stream (character oriented) and record (block oriented) files. Record files may be sequential or direct (random), keyed or not keyed; the length of a key is implementation-dependent [2, Section 1.2.2 (25)].

c. The ENVIRONMENT attribute allows the programmer to define machine-dependent features in an implementation-dependent manner [2, Section 1.2.2 (5) and (2)]. The syntax, semantics, and any consistency checks of the environment attribute are entirely implementation-dependent.

d. The special machine-dependent features of peripheral equipment are localized in the environment attribute, and the I/O capabilities of PL/I are general enough to handle the special cases.

e. Definition of completely machine independent interfaces to special machine features is now beyond the state-of-the-art. For PL/I, it would require definition of the entire operating system.

## Section VIII. CONTROL STRUCTURES

1. Kinds of Control Structures (G1).

a. Degree of compliance: P

b. PL/I provides structured control mechanisms for sequential, conditional, iterative, and recursive control, and for exception handling. PL/I does not provide control structures for (pseudo or real) parallel processing or asynchronous interrupt handling.

c. The PL/I "DO" allows multiple termination conditions [2, Section 3.1.10 A64 and A65], but these conditions are tested before each iteration [2, Section 6.3.4 and subsections]. Some iterative forms of the PL/I "DO" require no loop control variable [2, Section 3.1.10 A71].

d. The IBM versions of PL/I have a tasking or parallel processing facility. A similar capability was not included in the standard because it was difficult to define rigorously and it depended heavily on the operating system available. The same can be said of language features to handle asynchronous interrupts. These features would be difficult to define and implement and would substantially increase the size and complexity of an implementation.

2. The GoTo (G2).

a. Degree of compliance: F

b. PL/I provides a "GoTo" operation applicable to, but not restricted to, program labels within the most local scope of definition [2, Section 6.3.7]. PL/I imposes no unnecessay cost for the presence or use of "GoTo". Necessary costs are imposed for the orderly termination of all active blocks when the "GoTo" exits from one or more blocks [2, Section 6.3.7 Case 2.2].

c. PL/I offers label variables and arrays of label constants or label variables which may act as switches, and label parameters; numeric labels are not offered. The PL/I "GoTo" has very low security.

d. As in many other parts of PL/I, the major difficulty with "GoTo" is the provision of unwanted generality.

Removal of label variables and label parameters from the language, which could easily be achieved, would meet many of the goals of this Tinman requirement. More difficult would be the restriction of label namescope to a single level; this requires that treatment of labels be different from the treatment for other names, which could be a major change to identifer processing. Restrictions on the use of arrays of label constants (PL/I's version of computed GoTo) or provision of a case construct (see paragraph G3) would bring PL/I into full compliance with this requirement. It may be noted that PL/I does not have any non-normal loop exit except GoTo.

3. Conditional Control (G3).

a. Degree of compliance: P

b. PL/I conditional control structures are not fully partitioned. When there are nested IF-statements only some of which contain ELSE's, then an ELSE is associated with the "nearest" IF not having an ELSE.

c. The use of an array of label constants allows computed choice among labeled alternaives. This PL/I mechanism -- e.g., "GoTo CHOICES(N);" -- does not offer the clarity or security of the CASE construct.

d. The selection condition in an IF-statement may only be based on a Boolean expression. There is nothing quite comparable to Zahn's device offered by PL/I but the available control structures could be composed to provide a similar capability (though without the security offered by Zahn's device).

e. Restriction and modification of the GoTo and array of label constants mechanism could easily provide a CASE type conditional structure. To extend this CASE structure to discriminated union requires first a discriminated union capability, a large task which was discussed in paragraph A7 above.

f. Only a small change in syntax is necessary to require fully partitioned IF-THEN-ELSE; however, the possibly large number of do-nothing ELSE-clauses may detract from clarity.

4. Iterative Control (G4).

a. Degree of compliance: P

b. The PL/I iterative control structure, the "DO" loop, permits the termination condition to appear only at the beginning of the loop [2, Section 2.3.3 CH6 and Section 2.4.8 CM62]. The control variable is not local to the loop; moreover, there is no requirement that it be local to the block immediately containing the DO-loop. One form of the PL/I DO-loop, the "DO WHILE (expression)", requires no loop control variable. The PL/I loop "must" only be entered from the head of the loop [2, Section 6.3.7.1 Step 1]. (As with all requirements stated in [2], the action when a "must" constraint is violated is implementation-dependent.)

c. The PL/I DO-loop may take several different forms. In all the possible cases, the loop control variable has a well-defined value, but the multitude of cases means that a programmer would have great difficulty determining what that well-defined value is defined to be.

d. Any computed value may be passed outside the loop by assignment to a variable.

e. The addition of a loop exit command that can appear in a THEN or ELSE-clause, a minor addition, would permit the loop termination condition to appear anywhere; GOTO is used in current PL/I compilers for the same purpose. The restrictions on GOTO discussed under paragraph G2 above would allow compile-time checks against jumping into a loop body which is an implementation problem of some difficulty. Restricting the scope of the loop control variable would require some changes to the structure of DO-loops and declarations; the loop control must be implicitly declared and the loop treated as a scope. This is not a minor change.

5. Routines (G5).

a. Degree of compliance: PT

b. PL/I offers recursive routines as a programmer-specified option. There are no restrictions on defining procedures within recursive procedures; including such restrictions is not a major change.

6. Parallel Processing (G6).

a. Degree of compliance: P

b. PL/I offers no parallel processing capability (although such facilities have been included in previous versions of the language).

c. The inclusion of a parallel processing facility in PL/I would be difficult in both the definition and the implementation phases. As stated by Marcotty [15, p. 133]:

> During the course of the development of the language for standardization, a tasking facility was designed and it was only during the final stages of preparation of the definition document that the facility was removed. The main reasons for this action were difficulties with the rigorous definition of the facilities and their interaction with storage. There was also a feeling that with current operating systems, there was no great advantage to be obtained by multitasking within a program. I believe it is currently beyond the state of the art to provide a rigorously defined tasking facility that can be implemented securely and efficiently.

7. Exception Handling (G7).

a. Degree of compliance: P

b. The PL/I exception handling control structure, the on-unit, allows the user to enable transfer of control for any error or exception situation recognized by the system. Aside from conversion errors (e.g., the attempt to convert non-numeric character data to a numeric value) and certain I/O errors, the exception handling mechanism is not parameterized. The exception handling process has no way of determining the source of the exception (i.e., the statement whose execution caused the exception) and, except for conversion errors, no way of correcting the situation that caused the exception.

c. A PL/I exception causes control to transfer backward because a condition is enabled where the procedure for handling the exception is defined. For every exception

condition not explicitly enabled there is a system exception procedure -- implementation defined. The PL/I user can write programs which can escape from an arbitrary nest of control and the exception handling procedure can be redefined for different scopes and for different portions of the same scope.

d. PL/I also provides the user with a mechanism to define his own named conditions which are activated through use of SIGNAL statement [2, Section 6.4.1.3] (e.g., IF I>50 THEN SIGNAL INDEX_TOO_LARGE; where INDEX_TOO_LARGE is a programmer-named condition, which is defined by the programmer-written procedure handling it).

e. While PL/I meets much of this requirement, full compliance would involve great expense and a new definition of this language feature. For instance, parameterizing the recovery requires that all possible exception points and exception types have some mechanism for using this recovery information. To meet the partial requirement that transfers of control be forward in the program would likely result in a facility which is less clear and less useful than the one presently in PL/I.

8. Synchronization and Real Time (G8).

a. Degree of compliance: P

b. PL/I offers no real time or parallel processing facilities; thus, no synchronization features are available. Addition of such a capability to PL/I would involve great expense; see the discussion in connection with G6 above. (It may be noted that previous versions and implementations of PL/I have contained synchronization and real-time facilities which meet some of the requirements of G8.)

## Section IX. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics (H1).

a. Degree of compliance: P

b. The PL/I source language uses free format with an explicit statement terminator [2, Section 2.4 Middle Level Syntax]. PL/I allows the use of mnemonically significant identifiers. Implementations must allow identifiers up to 31 characters in length and may allow larger identifiers. PL/I is based on conventional forms. PL/I does not have a simple grammar. The very large number of keywords (188, which are not reserved), the number of optional forms for nearly every statement type, and the complexity of certain constructs make the language difficult to parse. There are also 87 built-in functions whose names should, in a disciplined programming environment, be reserved unless the user's explicit intention is to replace the built-in function with a programmer-written function serving the same purpose. Abbreviation of many of the keywords is permitted.

c. Although it is certainly possible to write readable, maintainable programs in PL/I, many aspects of the language design foil this goal. One of the best illustrations of this is the policy of trying to construe a legitimate interpretation for programs which are almost certainly in error. For example, the fragment "DO I=Ø, I=1; loop body; END" will execute the loop body twice, but each time with the control variable (I) equal to zero because the "I=1" is interpreted as a Boolean expression.

d. Some of the simplifications for PL/I discussed elsewhere will make programs written in the language easier to read and to compile. The removal of some or all of the unneeded features listed in Section XV would also tend to make PL/I a simpler language. Abbreviated keywords in PL/I are considered fully equivalent to the unabbreviated keywords; based on frequency of usage or some other criterion, only one of the forms should be used. This change would remove 45 keywords, some very unclear (e.g., NOUFL for NOUNDERFLOW or CTL for CONTROLLED), from the language. Reserving particular symbols and operators to a single purpose, as discussed under paragraph H10 below, will also remove much ambiguity and error-proneness from PL/I. All of these changes would simplify new implementations of

PL/I. The changes to existing compilers, while individually quite easy to design and implement, would involve substantial effort because of the multitude of changes required.

2. No Syntax Extension (H2).

a. Degree of compliance: T

b. PL/I offers none of the interdicted features.

3. Source Character Set (H3).

a. Degree of compliance: PT

b. The only PL/I characters not in the 64-character USASCII subset are the "not"-symbol and the "|" (logical "or" and string concatenate) symbols. The language definition does not specify a translation of those two characters.

c. The original versions of PL/I were defined to use the EBCDIC character subset available on keypunches. The two characters above could be translated to circumflex and exclamation point, which have no other PL/I meanings, in the 64 character USASCII subset. These are minor changes.

4. Identifiers and Literals (H4).

a. Degree of compliance: P

b. The PL/I language definition provides formation rules for identifiers [2, Section 2.5.3 CL8] and literals [2, Sections 2.5.4 and 2.5.5]. The identifiers may contain an explicit break character (the underline character). Neither break characters nor blanks may appear in literals. PL/I does not require (or allow) a separate quoting of each line of a long literal.

c. The formation rules for literals could probably be modified to accept (i.e., ignore) embedded break characters. A somewhat larger change would be involved in allowing the separate quoting of long literals. It may be noted that if constant expresions are evaluated at compile time (see paragraph C4 above) then the concatenate operator can be used between the quoted literals with no change to the language syntax.

5. Lexical Units and Lines (H5).

a. Degree of compliance: P

b. PL/I does not treat end-of-line as a delimiter [2, Section 2.5.1 CL3]; thus, lexical units may continue across line boundaries. End-of-line may be included in string literals [2, Section 2.5.5 CL25].

c. Identifiers and strings may not consist of more than one lexical unit nor are constant expressions necessarily evaluated at compile-time.

d. PL/I could easily be changed so that lexical units are not allowed to continue across lines. Constant expressions could be evaluated before run-time. PL/I could only with difficulty allow identifiers to consist of more than one lexical unit.

6. Key Words (H6).

a. Degree of compliance: P

b. PL/I keywords are not reserved (hence are usable as identifiers), are very numerous, and are generally informative. The names of built-in functions of PL/I are treated as identifiers. Program specified condition names and the built-in conditions may appear in the same contexts.

c. Elimination of abbreviations would remove some 45 keywords (i.e., more keywords than exist in FORTRAN) from PL/I. Some keywords could be removed if unneeded features were removed (see Section XV). The remaining keywords of PL/I could be made reserved words in the language. PL/I must continue to have a large number of keywords simply to name the plethora of possible identifier attributes (currently 49 keywords) and exception conditions (currently 29 keywords). Achieving PL/I's goals with only a small set of keywords would be extremely difficult, but the keywords that exist could easily be reserved.

7. Comment Conventions (H7).

a. Degree of compliance: PT

b. PL/I has a single uniform comment convention [15, Section 2.5.2]. A PL/I comment is opened by "/*", closed by "*/". A comment may appear anywhere a blank may appear (i.e., not internally to a lexical unit). A PL/I comment does not terminate at an end of line; only the comment closing symbol terminates a comment.

c. A currently unused character (e.g., "$") could be used to begin a comment terminated by end of line, which would allow PL/I to meet this Tinman requirement without difficulty.

8. Unmatched Parentheses (H8).

a. Degree of compliance: F

b. PL/I allows more BEGINs than matching ENDs and more DOs than matching ENDs, but only when particularly sought by the programmer.

c. The PL/I convention allows a named END statement to close all unclosed BEGIN blocks and DO groups contained within the BEGIN block or DO group with the same name. Elimination of this convention from PL/I would result in no lessened capability in the language, could be easily carried out, and would simplify new compilers.

9. Uniform Referent Notation (H9).

a. Degree of compliance: P

b. PL/I partially satisfies this requirement, since array referencing and function invocations share some similar syntactic forms. However, the "*" form of subscript may not be used as a function argument; except for the "pseudovariables", function invocations cannot appear on the left side of assignment statements while array references can; the dot qualification form for STRUCTURE component selection cannot be used in a function call.

c. Providing an entirely uniform referent notation presents an extremely difficult definitional problem. What would a "*" form of procedure argument mean? What does a function as target of an assignment statement mean? The only approach likely to be successful is the restriction of certain capabilities in the language.

10. Consistency of Meaning (H10).

a. Degree of compliance: P

b. PL/I does allow language defined symbols to have different meanings in different contexts as follows:

(1) Colon ":" delimits statement labels and condition prefixes and separates lower and upper bounds in array dimension declarations.

(2) Period "." serves as the decimal point in literals and a qualifier delimiter for STRUCTURE component selection.

(3) Asterisk "*" is the multiply operator and can stand for an unknown string length or free array bound.

(4) The keyword ERROR can indicate a compile-time attribute error in the DEFAULT statement or serve as the run time exception condition "of-last-resort".

(5) Parentheses "()" either enclose lists or specify precedence.

(6) The equal sign "=" is the assignment operator and a relational operator.

c. Only the last two of the above exceptions to this Tinman requirement are ever likely to lead to ambiguity in practice; item (6) is especially bad (e.g., "I=I=1;" does not have an obvious meaning; see paragraph H1 above.). A new operator for assignment (possibly ":=" which is commonly used for assignment) could be defined. The different functions of parentheses in (5) above result in the different meanings of "()" in CALL PROC(A, (B), C); assuming argument and parameter types match, A and C are passed by reference and B is passed by value. The binding of argument to formal parameter should be specified by the procedure called rather than by the caller. See paragraph C7 above.

## Section X. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic (I1).

a. Degree of compliance: F

b. PL/I implementation may define many parameters: the default and maximum precisions of variables, the size of a data record on an external medium, and many others. An implementation may also define when exception conditions are raised.

c. The behavior of a program which does not conform to the standard is entirely implementation-dependent [2, Section 1.2 ("Relationships between an implementation and This Definition")]:

> An implementation's interpretation of a program-run conforms to the standard if and only if it conforms to one of the conceptual interpretations as follows:
> (1) If the conceptual interpretation rejects a program run (via failure of a "must" test) or if it never completes the translation-phase, then any interpretation by the implementation conforms. In particular, an implementation may or may not reject a program-run at the same point as it is rejected by the standard, or at all.
> (2) Otherwise, the implementation's interpretation conforms if it makes the same sequence of changes to datasets as does the conceptual interpretation.
> (3) The implementation's interpretation also conforms if it deviates from (1) and (2) only as permitted by the flexibilities of interpretation specified in Section 1.2.1.
> Note that this implies that an implementation may provide extensions beyond the language defined in this standard, but is still required to conform for a program not using those extensions just as if the extensions were not available.

d. The greatest weakness of PL/I with respect to this Tinman requirement involves the completely undefined

behavior of erroneous programs. In terms of the standard this problem could be solved by defining standandized behavior for certain classes of error. For example:

(1) All syntax errors will generate a message at the earliest possible point (which depends on the parsing technique and the grammar). An object program compiled from source text containing syntax errors shall be executable only up to the point of the error.

(2) There shall be a compiler/system option to detect and announce references to undefined variables.

(3) Jumps into the bodies of loops shall be detected and announced.

(4) Violation of any "must" constraint on the use of pointers shall be detected and announced.

e. Certain of the implementation-dependent features that remain after specifying the behavior of illegal programs are not violations of the Tinman (for instance, allowing the maximum precision to be implementation-dependent). The elimination of defaults throughout PL/I removes the implementation-dependencies in the default specifications. Taken together, the restrictions proposed above involve a definition phase and, depending on hardware support facilities available, a possibly very expensive implementation phase.

2. Object Representation Specifications Optional (I2).

a. Degree of compliance: P

b. PL/I partially satisfies this requirement. The programmer may specify alignment for data objects, thereby overriding the translator-defined default. However, there is no facility for specifying whether a subroutine is to be compiled open or closed. As no parallel processing is defined for PL/I, reentrant vs. nonreentrant is not an applicable option.

c. Although compile-time macro capabilities are not defined in the PL/I standard, many compilers for PL/I support such a feature, which allows a possible

implementation of open subroutines. At the expense of somewhat greater complexity, "open" vs. "closed" could be specified in the procedure heading.

3. Compile Time Variables (I3).

a. Degree of compliance: F

b. PL/I has no features for meeting this requirement.

c. A set of reserved words similar to the set used in JOVIAL (see Chapter 5) could be made available for compile-time and run-time interrogation. The JOVIAL set specifies: bits in a word, bits in a byte, maximum available memory size, etc. Bringing PL/I into compliance with this requirement is a relatively minor modification, both to language and implementation.

4. Conditional Compilation (I4).

a. Degree of compliance: F

b. PL/I provides no conditional compilation facilities.

c. Although not defined in the PL/I standard, a compile-time macro facility is provided in many existing implementations of PL/I. This facility was omitted from the standard because of an inability to agree on a suitably powerful capability and the difficulty of rigorous definition of the interaction with other PL/I capabilities. A facility adequate to meet this Tinman requirement could be defined and implemented at a reasonable cost.

5. Simple Base Language (I5).

a. Degree of compliance: F

b. PL/I provides no extension facilities, so the entire language must be considered the base. Some features of PL/I are redundant, for example: "DO v=e1 BY e2 TO e3;" is a simplication of "DO v=e1 REPEAT e2 WHILE (e3);". Some other features are so similar that it may be difficult to determine which is best for a particular application: CONTROLLED storage and BASED storage both have dynamic allocation under explicit user commands; CONTROLLED storage may be declared external and BASED storage may not; pointers and offsets also have many similarities.

c. It is not realistic to expect to modify PL/I to bring it into compliance with I5 and still have the language retain its essential structure. Pl/I was not designed to be a simple language.

6. Translator Restrictions (I6).

a. Degree of compliance: F

b. These restrictions are a function of the translator implementation [2, Section 1.2.1.2 (1) and (2)]. The language definition does not restrict the number of array dimensions or the number of arguments for a function or subroutine call; when an implementation sets a maximum length on identifiers, that maximum may not be less than 31 characters [2, Section 1.2.1.2 (1)]; the number of nested parentheses levels in expressions is not restricted nor is the number of identifiers.

c. It is a minor modification to add to the language definition a set of restrictions such as the ones called for in I6. The problem lies in deciding what these restrictions should be.

7. Object Machine Restrictions (I7).

a. Degree of compliance: T

b. PL/I satisfies this requirement. Any excessive static requirements for system resources may be announced at compile time or load time; any excessive dynamic requirements for system resources raise run-time exception conditions. No restrictions dependent on the object machine are built into the language definition.

## Section XI. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

1. Efficient Object Code (J1).

a. Degree of compliance: P

b. There are many PL/I features which require run-time support even when not used. Strings and arrays require dope vectors, even when they are of fixed length or have fixed dimensions, because argument passing allows run-time determination of parameter bounds. The use of array slices which PL/I allows also has a high run-time cost. Although not strictly a run-time cost, assigning all the defaults to PL/I variables makes the compilers more expensive and slower running. The large number of attributes and the complicated way they are assigned to literals, for example, means that unless a programmer has been extremely careful some implicit conversions will be involved. For example:

```
DECLARE X FLOAT;
X = X + 1.0; /* 1.0 REQUIRES CONVERSION FROM
FIXED TO FLOAT*/
X = X + 1.0E0; /* REQUIRES NO CONVERSION */
```

While this could be optimized, it would require additional overhead for the compiler.

c. The very richness of PL/I encourages programmers to use inefficient features which they do not need and might otherwise avoid. An example is the complicated processing allowed on records (i.e., PL/I structures) of dissimilar form:

```
DECLARE 1 A,
          2 B,
            3 C FLOAT,
            3 D FLOAT,
          2 E FLOAT;
DECLARE  1 Z,
          2 Y(3) FLOAT,
          2 X    FLOAT;
```

The operation A + Z results in an intermediate structure of the following form:

```
1 tmp,
   2 B_Y(3),
      3 C FLOAT,
      3 D FLOAT,
   2 E_X FLOAT;
```

The values are:

```
tmp.B_Y(1).C = A.B.C + Z.Y(1);
tmp.B_Y(1).D = A.B.D + Z.Y(1);
tmp.B_Y(2).C = A.B.C + Z.Y(2);
tmp.B_Y(2).D = A.B.D + Z.Y(2);
tmp.B_Y(3).C = A.B.C + Z.Y(3);
tmp.B_Y(3).D = A.B.D + Z.Y(3);
tmp.E_X = A.E + Z.X;
```

d. The PL/I standard [2, Section 6.3.7.1 Step 1] states one "must not" GOTO a statement within a loop. Because PL/I allows label variables, the compiler cannot check that this error will not occur. Either a high run-time overhead, both of time and information, is required, or the error is ignored.

e. Changes suggested elsewhere in this chapter will contribute to the goal of efficient object code by eliminating implicit conversions, by eliminating the promoting of structures to different forms, and by removing the need for certain run time checks by restricting labels, GOTO and pointer variables. Were all such changes included, PL/I efficiency would compare favorably with other high order languages. However, these are fairly substantial modifications, both to the language and to implementations.

2. Optimizations Do Not Change Program Effect (J2).

a. Degree of compliance: F

b. The PL/I standard explicitly allows certain side effects to be ignored [2, Section 1.2.7.4 "Flexibilities of Interpretation: Expression Evaluation"]. The order of evaluation of contained subexpressions may be changed, which may change the order of evaluation of function subprograms. Any optimizations performed within the language specification could change the effect of a program.

c. The great flexibility allowed by the PL/I standard is explicitly intended to allow compiler optimizations. A restriction that all functions in an expression be evaluated in a left-to-right order reduces the possible optimizations. Certain optimizing compilers would be simpler because of this restriction; nonoptimizing compilers would not be affected. Because of PL/I exception condition processing, a requirement that <u>all</u> side effects occur in left-to-right

order would, for practical purposes, eliminate the possibility of optimization.

3. Machine Language Insertions (J3).

a. Degree of compliance: F

b. PL/I offers no embedded machine language code capability.

c. Implemented PL/I compiler systems allow any procedure, including machine language procedures, with compatible calling sequences to be included in a complete PL/I program. This approach (not formalized in [2]) offers several advantages over embedded assembler code. The interface to the procedure is well-defined, and on another machine another procedure with the same interface could be written; the machine language procedure, once written and tested, could be included in a library and shared by many users, which is not so easy for embedded code; the interface to PL/I is narrower; and PL/I would not need a full target machine assembler to compile programs.

d. If machine language subroutines are not adequate to meet this requirement then a substantial extension to PL/I is necessary. The access rights of the machine language code to the PL/I declared variables, the preservation and restoration of processor states at the entry and exit points of the machine language code and many other such points must be defined. Moreover, a full or nearly full assembly capability must be added to the already large PL/I compiler.

4. Object Representation Specifications (J4).

a. Degree of compliance: P

b. PL/I partially meets this requirement for records by allowing an UNALIGNED attribute (UNALIGNED data are packed as densely as possible). It is possible to specify the order of fields (order in the DECLARE statement), the width of fields, and the presence of "don't care" fields. The ALIGNED attribute will force data to a word boundary. A programmer may specify the precision to minimize the storage required.

c. The PL/I programmer may not explicitly control the object representation of a composite data structure; however, it is possible to tailor alignments and precisions to a particular object machine and so remove portability in a quite insidious fashion.

d. Use of the "compile time variables" required in I3, together with existing features of PL/I such as arbitrary length bit strings and DEFINED data, are adequate to meet this requirement. Something similar to the more sophisticated and explicit strategy used in the JOVIAL specified table would involve a large design and implementation effort.

5. Open and Closed Subroutine Calls (J5).

a. Degree of compliance: F

b. PL/I does not provide the user the option of specifying open vs. closed subroutine invocations.

c. Some PL/I implementations offer a compile-time macro facility, which achieves one form of open subroutine. Defining a procedure to be "open" in the procedure heading would involve extensions to PL/I. The effect on the language is minor, but impact on implementation may not be trivial.

## Section XII. PROGRAM ENVIRONMENT

1. Operating System Not Required (K1).

a. Degree of compliance: PU

b. While PL/I without question requires the presence of an operating system, the standard uses the hardware/operating system combination as an abstract machine. The standard defines an abstract interpretive PL/I machine and does not distinguish among operating system capabilities, run time support capabilities, and those capabilities compiled directly into the object source code. Substantial operating system capabilities are required to serve the wide I/O capability of PL/I and to provide the memory management required for automatic, based, static, and controlled storage. In a typical PL/I system many of the exception conditions must be processed by the operating system before control passes to the user exception condition process.

c. It is not feasible to bring PL/I into compliance with K1 without thereby conflicting with other Tinman requirements (e.g., B10, G6, G7, G8).

2. Program Assembly (K2).

a. Degree of compliance: PU

b. The PL/I standard permits a program to be composed of separate procedures [2, Section 1.4.3.3 and Section 3.1.1]. The compilation and linking of these separate procedures is described only interpretively and abstractly. The standard validates all procedure interfaces as though the program were produced in a single compilation.

c. Existing PL/I compilers and systems provide separate compilation and libraries, and fully support integration of separately written modules into an operational program. Existing systems generally offer an inadequate check of the interface between separately compiled modules. Since the linker, which should provide the validation, is common to many different compilers, it would be difficult to improve this validation without great expense.

3. Software Development Tools (K3).

a. Degree of compliance: U

b. The PL/I standard defines only the abstract interpretive machine. There is no mention whatever of the desirability of listings, cross references, linkers or any type of debugging system.

c. Existing PL/I implementations provide many of the Tinman-required software development tools, although there is no user interface which is consistent from system to system. An implementer's guide that described, more informally than the PL/I standard, a suggested programming environment would be extremely useful. Possible topics include: listing format and information provided; the format of error messages and when they should appear; and the debugging information which should be available.

4. Translator Options (K4).

a. Degree of compliance: U

b. Comments under paragraph K3 above apply here as well.

5. Assertions and Other Optional Specifications (K5).

a. Degree of compliance: PU

b. Any text having the status of comments can be included as comments. PL/I offers no assertion feature; however, the conditional and output (e.g., PUT DATA) could be used to provide a substitute. If there is a compile-time macro facility available, then nearly all the requested flexibility would become available.

## Section XIII. TRANSLATORS

1. No Superset Implementations (L1).

a. Degree of compliance: FU

b. Supersets are explicitly permitted in [2, Section 1.2]: "Note that this implies that an implementation may provide extensions beyond the language defined in this standard, but is still required to conform for a program not using those extensions just as if the extensions were not available."

c. The PL/I standards committee also has a subcommittee investigating a real time version of PL/I. The intention is to design a set of real time tasking features that can be well defined and then add those features to the full language standard during the coming standardization review cycle.

d. The restriction of supersets could be easily incorporated into the PL/I standard definition; however, it is questionable whether supersets would disappear under such pressure. A more effective approach might be to require a compiler option to list all deviations from strict standard conformance; such an option would be straightforward to implement.

2. No Subset Implementations (L2).

a. Degree of compliance: U

b. The PL/I standard has no provision for subsets.

c. PL/I became a USANSI standard language on 9 August 1976, so there are as yet no fully standard versions of PL/I implemented. The size and complexity of PL/I are such that only large machines can support full PL/I systems; thus, it is likely that subset languages will appear. In fact, the PL/I standards committee has formed a subcommittee to define a general purpose subset of PL/I suitable for standardization.

3. Low Cost Translation (L3).

a. Degree of compliance: FU

b. Low cost translation is a feature of the compiler and the system as well as the language. PL/I has a number of difficult-to-compile features, such as the default declaration scheme, many unreserved keywords, and operations on STRUCTURE operands. Significant redesign is needed if low cost translation is to be a serious objective of PL/I.

4. Many Object Machines (L4).

a. Degree of compliance: U

b. This is a management consideration and is not addressed in the language definition.

5. Self-Hosting Not Required (L5).

a. Degree of compliance: U

b. This requirement is a mangement consideration and is not addressed in the language definition.

6. Translator Checking Required (L6).

a. Degree of compliance: U

b. The PL/I standard explicitly makes all error checking dependent upon implementation. As stated in [2, Section 1.2]:

> An implementation's interpretation of a program-run conforms to the standard if and only if it conforms to one of the conceptual interpretations as follows.... If the conceptual interpretation rejects a program-run (via failure of a "must" test) or if it never completes the translation phase, then any interpretation by the implementation conforms. In particular, an implementation may or may not reject a program-run at the same point as it is rejected by the standard, or at all.

c. The PL/I standard could specify classes of errors which are to be detected and, for each class, when they are

to be detected and what is to be done upon detection. This modification would require a significant amount of effort.

7. Diagnostic Messages (L7).

a. Degree of compliance: U

b. The PL/I standard provides no specification or suggestion of useful diagnostic or warning messages. As noted under L6 above, detection and announcement of errors of any type is entirely implementation dependent.

8. Translator Internal Structure (L8).

a. Degree of compliance: T

b. The PL/I standard nowhere specifies what the internal structure of a translator should be.

9. Self-Implementable Language (L9).

a. Degree of compliance: U

b. This requirement involves management considerations not addressed in the standard. PL/I has all capabilities needed to write a compiler. The current lack of type checking is the major shortcoming.

## Section XIV. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only (M1).

a. Degree of compliance: P

b. As an existing commercial language PL/I uses only features within the state of the art. Some research would be needed to achieve the Tinman's needed characteristics. This is especially true for the parallel processing and data abstraction capabilities.

2. Unambiguous Definition (M2).

a. Degree of compliance: P

b. The PL/I standard defines the language using a simplification of the Vienna Definition Language. This simplification is a stylized English prose type of programming language, which attempts to be formal and rigorous while remaining more comprehensible than the Vienna Definition Language. Most of [2] can be understood by anyone with a good working knowledge of high-order languages; parts of it are extremely confusing and some ambiguities and "bugs" remain in the document. The committee has improved, and continues to improve, the definition by recomposing the confusing portions and removing ambiguities and bugs. A major flaw in [2] is the total absence of specification for the processing of illegal programs.

c. The PL/I standard comprises the folowing sections: the concrete syntax; the abstract syntax; the translator; the PL/I interpreter (i.e., the PL/I machine); and the flow of control and action routines for the PL/I machine. The concrete syntax describes the form the input program text must take. Thus, all keywords and delimiters are defined in the concrete syntax. The translator describes the translation of the concrete syntax into the abstract syntax, which reflects the concrete syntax but has all the identifier attribute lists completed using defaults, and the entire program represented in a standardized tree form with no keywords or delimiters. The PL/I interpreter describes an idealized processor which runs programs in the tree format of the abstract syntax. This processor has no speed or storage constraints. The flow of control and action

routines describe interpretively, in a stylized English prose, the processing of each PL/I unit from the procedure entry and exit actions to the single expression actions.

1. The PL/I standard is intended for the language implementor rather than for the PL/I user. The user should know the implementation-dependencies as well as any non-conforming parts defined by his implementation before using a PL/I implementation.

3. Language Documentation (M3).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

c. All implementations of PL/I have a set of documentation ranging from the introductory level to quite formal descriptions of the language. Also generally described are the operating system interfaces and other implementation-dependencies. PL/I is used in a number of text books on introductory programming, which provide a machine-independent guide to the language.

4. Control Agent Required (M4).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

5. Support Agent Required (M5).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

6. Library Standards and Support Required (M6).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

## Section XV. CONCLUSIONS REGARDING PL/I

### 1. Conflicts Between the Objectives of PL/I and the Tinman.

a. The major conflicts between the Tinman and PL/I arise because of differences between their respective design goals. Many Tinman requirements involve security and reliability. PL/I, however, was designed primarily for power and for wide applicability. One of the major design goals of PL/I was to include sufficient power and generality to allow any program written in either FORTRAN or COBOL to be better written in PL/I. Modularity (i.e., that a user need know only those PL/I features directly applicable to his problem) and commonality of features with COBOL and FORTRAN were also goals. PL/I includes features from COBOL, FORTRAN, JOVIAL and ALGOL; opinion differs on how well the features of these disparate languages have been integrated into a consistent whole. PL/I is designed to facilitate the initial writing of a program, at the expense of ease in debugging and maintenance. Examples abound: the non-mnemonic abbreviations for exception conditions (e.g., NOUFL for no UNDERFLOW) that will likely occur only once in programs; the labelled end that closes multiple blocks, which even in very large programs cannot save significant effort, and, of course, the defaults and the implicit conversions. PL/I lacks any generalized extensibility and thus the base language supporting all its many features is huge. New features can be included in the base language, at the cost of increased size and complexity; nonetheless, this is the general approach taken by proponents of PL/I. This is understandable in view of the eclectic nature of the current PL/I language.

b. The goals of the Tinman clearly differ those of PL/I. Although power is important in a Tinman-type language, this is secondary to reliability and security. The Tinman favors ease of program debugging and maintenance over ease of writing. It may be noted that many of the Tinman objectives are the direct result of adverse experience in the use of PL/I; thus, unrestricted power in a language is not likely to be a successful approach.

### 2. Summary of Major Areas of Conflict Between PL/I and Tinman.

a. Data and Types. Although PL/I partially satisfies each of these requirements, PL/I applies much weaker restrictions on defaults and declarations than those the Tinman would apply. Of the required features, PL/I lacks (1) a discriminated union facility, and (2) explicit, namescope-level specification of floating point precision.

b. Operations. The major conflict here is that PL/I allows implicit conversions from any computational mode to any other and that it allows operations on non-conformable records. PL/I also lacks enumeration types and power sets, but many of the effects of these operations may be achieved with bit strings. Certain fixed point operations (e.g., division) in PL/I produce effects somewhat different from those required by the Tinman.

c. Expressions and Parameters. PL/I has a number of serious conflicts with the Tinman in this area. Side effects in PL/I are not constrained to occur in a left-to-right order. The PL/I language definition allows, but does not require, the compile-time evaluation of constant expressions. PL/I does not effectively enforce type agreement in parameters, nor does it have the capability to specify binding class on the formal parameter side. A fixed number of arguments to procedures is required. Most of these conflicts result from PL/I's relative lack of constraints and security enforcement.

d. Variables, Literals and Constants. PL/I has two major conflicts with these Tinman requirements: PL/I has no range restrictions on variables, and the pointer mechanism is not secure. With both of these requirements the changes necessary for conformance to the Tinman are difficult to define well and would involve great expense in implementation and again in run-time overhead. PL/I's other shortcomings in this area could be removed fairly easily. Constant value identifiers could be added to the language; the specifications for numeric literals and input data and their relationship could be made more explicit. Without hardware support it is difficult (i.e., very expensive at run-time) to determine if variables have been initialized.

e. Definition Facilities. In this area, PL/I conflicts with the Tinman in that (1) PL/I lacks any encapsulated data type definition capability, (2) PL/I allows defaults, (3) PL/I does not provide data definition through enumeration of

literal names, and (4) PL/I does not provide for operator extension or definition. Enumeration types could be included in PL/I without great difficulty but the true type definition and data abstration capability which the Tinman seeks would involve a major extension to the language.

f. Scopes and Libraries. Aside from features altogether missing from PL/I because it lacks type definition capabilities, PL/I has no major conflicts with Tinman requirements in this area. The nature of PL/I's defining document is such that many "real world" characteristics, such as libraries, are not really described. The PL/I standard describes the behavior of a program comprising standard conforming procedures irrespective of the source of those procedures. All PL/I systems have libraries but the nature and use of such libraries are not defined in the PL/I standard. The use of %INCLUDE, which inserts source code into the program, is not so secure as a COMPOOL, which uses partially or fully compiled programs. This is because the attributes of identifiers in %INCLUDEd text can depend on the surrounding source (i.e., can vary from procedure to procedure), whereas the attributes of identifiers in a COMPOOL are fixed for all invocations of the COMPOOL.

g. Control Structures. PL/I conflicts with these Tinman requirements primarily in that PL/I lacks parallel processing facilities and has an unrestricted GOTO. Minor conflicts are PL/I's lack of a CASE-type conditional construct and its lack of a loop termination condition that can appear anywhere in a loop.

h. Syntax and Comment Conventions. The single major conflict in this area involves uniform referent notation. PL/I's very large number of keywords also presents difficulties, but the simplifications suggested throughout this chapter will be helpful here.

i. Defaults, Conditional Compilation and Language Restrictions. PL/I has neither provisions for parameterizing the object machine nor any facility for conditional compilation. The most serious conflict with these Tinman requirements lies in PL/I's leaving all error detection and processing entirely implementation-dependent.

j. Efficient Object Representations and Machine Dependencies. PL/I has several major conflicts with Tinman

requirements in this area: PL/I does not contribute effectively to a programmer's writing efficient code and it contains many very inefficient constructs. The specifically unspecified order of expression evaluation means that optimzations can change program effect; machine language insertins are not provided; and the user has no option for specifying open or closed subroutine calls.

k. Program Environment. Some of these Tinman requirements are not language requiremnts per se but rather implementation requirements or possibly separate products (e.g., Software Development Tools (K3) and Translator Options (K4)), which can be specified and purchased by DoD. As for language requrements per se, the single conflict between PL/I and the Tinman in this area is that PL/I lacks an assertion capability.

l. Translators. PL/I has major conflicts with these Tinman requirements because the processing of erroneous or non-standard conforming programs is not specified. Supersets are explicitly allowed in the standard. Some of the requirements in this area are not directly applicable to a language.

m. Language Definition, Standards and Control. These characteristics do not pertain to the language per se; it is not applicable to speak of conflicts between the published PL/I standard and these Tinman requirements.

3. Unnecessary Features in PL/I.

a. Introduction. One of the objectives of this report is to identify language features which are not needed to satisfy (but do not conflict with) the Tinman requirements, with a recomendation as to whether such features should be kept or eliminated. PL/I has a number of features which fall into this category. The following brief sections describe these unnecessary features and make appropriate recomendations.

b. BY NAME OPTION. This PL/I feature is carried over from COBOL's Move CORRESPONDING. The statement "A=B, BY NAME;", where A and B are two dissimilar STRUCTUREs with some component names in common (these components need not have the same attributes), will cause the component from "B" to be stored, after appropriate implicit conversions, in the

component of A that has the same component names. The BY NAME feature is costly to implement, does not appreciably increase the capability of the language, and is error-prone to write and difficult to maintain; we recommend its elimination.

c. Picture Data. This is another carry-over from COBOL. A "picture" represents a numeric data field maintained in character form with certain associated formatting information such as currency indicators. Computation with picture variables is very slow; PL/I has very flexible edited input and output which obviates the need for picture data. For these reasons we recommend the elimination of picture data from PL/I.

d. LIKE-Attribute. Use of the LIKE-attribute (in a STRUCTURE declaration) instructs the compiler to complete the declaration copying the component identifiers and attributes declared in the named STRUCTURE. Because the LIKE-attribute mirrors exactly the user's intentions and contributes to ease of maintenance by requiring that changes be made in only one place to affect all occurences of the same construct, we recommend its retention. Note also that all processing for the LIKE-attribute occurs at compile-time and that it use assures that STRUCTUREs to be used together are conformable.

e. CONTROLLED Storage Allocation. This storage allocation class provides the user with a stack of instances of the data object That is, when a CONTROLLED variable (or aggregate) is ALLOCATED the latest active instance of that variable is pushed and a new top-of-stack is available (and initialized if so declared). When the variable is FREEd then the previous instance is again available. The user could easily simulate this mechanism using based storage or a vector and an index. Because the presence of CONTROLLED storage complicates the language and the memory management system without providing any unique capability, we recommend its elimination.

f. Stream I/O. PL/I has three types of stream I/O: GET/PUT DATA; GET/PUT LIST; and GET/PUT EDIT. GET/PUT EDIT allows the user very explicit and detailed control of the format of the data read or written. For many applications such control is necessary. The LIST and DATA I/O are much simpler to use because the compiler selects the appropriate

formats. However, these two forms overlap. While neither GET/PUT DATA nor GET/PUT LIST are required by the Tinman, we recommend the retention of GET/PUT LIST and the elimination of GET/PUT DATA. GET/PUT LIST provides adequate capability and is much easier to implement.

4. Recommendations Concerning PL/I.

a. On the basis of the evaluation conducted in this chapter, we conclude that an attempt to achieve full compliance of PL/I with the Tinman requirements would necessitate changes so large as to result in a substantially different language. This is because the design objectives of a Tinman-like language and PL/I are so different and also because PL/I was designed before the utility of the requirements captured in the Tinman was known.

b. Some changes that could be easily made to PL/I would improve its performance with regard to Tinman requirements without affecting its power but change the "flavor" of the language. The removal of default declarations and implicit conversions from the language would simplify PL/I and result in better written programs, but would no longer give some meaning to all possibly valid constructs. A PL/I which was changed only by the restrictions described elsewhere in this chapter, would retain the advantages of using an existing language and be a significant improvement over current PL/I.

c. The addition of most Tinman-required features to PL/I would be difficult to do cleanly, would result in a larger and more complex language, and would involve a design, implementation and documentation effort greater than that required for a new language. The language so modified would have little in common with PL/I.

# CHAPTER 9

# SUMMARY

## Section I. EVALUATION TOOL

### 1. Technical and Overall Evaluation.

One of the objectives of this report is the derivation of a tool which can be used to provide information for the HOL selection process. The tool proposed is a methodology carried out in two stages: a Technical Evaluation (TE) followed by an Overall Evaluation (OE). The TE stage is a conceptually straightforward quantitative approach which is based on the separation of language-dependent and application-dependent elements. The OE stage takes into account factors which are specific to the computing environment in which the language is to be used. This is accomplished via a set of management decision-making criteria, whose values affect the various kinds of costs which comprise the expense for developing a software system. One of these criteria -- in fact, the most important one from the perspective of a language-procurer intending to implement large and long-lived software systems -- is technical merit, the score for which results from the TE stage.

### 2. Importance of Rating Matrix.

A significant feature of the Technical Evaluation stage is that the central component, the Rating Matrix of language features vs. goals, is independent of both language and application area. Thus, it need be derived only once, rather than for each language. Since this matrix contains almost 500 entries, the fact that it can be used repeatedly facilitates the language evaluation process.

### 3. Directions for Further Work.

As described in Chapter 2, paragraph I.6, there are several limitations on the applicability of the evaluation tool. First, the very nature of the method (i.e., the attempt to quantify judgments which are essentially imprecise) implies that the numeric results derived will contain a certain amount of "noise". Further research into

techniques to increase the accuracy of such a numeric approach would be desirable to help overcome this limitation. Second, the Overall Evaluation stage tends not to be directly applicable when the language considered in the Technical Evaluation stage differs from actual implementations -- e.g., a new version of an existing language. It is possible that work directed to the issue of estimating the "distance" between languages would be helpful here. Third, the evaluation tool does not reflect the issue of modifying the language to bring it into compliance with the identified requirements. Again, further research is needed in order to so extend the evaluation tool.

## Section II. TINMAN REVIEW

### 1. The "Tinman" as a Requirements Document.

It is important to recognize, both in reviewing the Tinman and in analyzing candidate HOLs with respect to its "needed characteristics", that the Tinman is intended as a specification of language requirements and not as a language design document. For this reason, the presence of conflicts between requirements is not surprising; in fact, it may reflect the state of affairs in the "real world" where, say, efficiency and reliability are uncompromisable objectives. On the other hand, a requirements document must be careful to avoid overspecification. Although the distinction between a requirement and a particular feature which meets the requirement is sometimes subtle, a requirement should not be specified to such a level of detail that solutions which satisfy the intent of the characteristic are ruled out. Unfortunately, the Tinman on occasion falls into this trap (e.g., regarding the pointer mechanism), as cited in Appendix III of this report.

### 2. Summary of Recommendations regarding the "Tinman".

Detailed comments on the individual characteristics appear in Appendix III. We emphasize here our agreement with the spirit of these requirements and are particularly pleased with the emphasis given to language constructs which promote sound programming methodology. The general comments below suggest changes which we feel would improve the overall quality of the document.

a. Reorganization. A problem with the current Tinman document is that some of the exposition is redundant (e.g., enumeration types arise in A5, B3, B11, D5, and E6). This has the disadvantage that a language lacking such a facility will be penalized several times (during the evaluation) for essentially the same reason. A somewhat different problem with the document is that a single characteristic sometimes covers a variety of requirements (e.g., G1, which includes sub-requirements for an assortment of control structures). This, too, makes the evaluation of a HOL more difficult, since the language may comply with the sub-requirements in varying degrees. Another problem with the Tinman document lies in the occasionally inappropriate section names; e.g., A7, titled "Records", really deals with Discriminated Union

Types; G5, titled "Routines", is concerned with recursion; I1, titled "No Defaults in Program Logic", is basically concerned with implementation dependencies as opposed to defaults.

b. Priorities. Because of the potential and sometimes actual conflicts among the Tinman requirements, it is important for a reader of the document to understand the priorities of the various characteristics. Clearly, the design of a HOL is based on decisions in which tradeoffs were made between conflicting objectives, and it is necessary in evaluating a language against the Tinman to keep in mind the priorities determined by DoD for the requirements of its common language(s). We recommend that such priorities be established, perhaps in the form of weightings for the various characteristics. A possible approach to this subject is provided in the Evaluation Tool described in Chapter 2: the product of the Rating Matrix and Application Vector is a vector whose elements serve, in effect, as weights for the Tinman requirements. (It might be profitable, in view of the wide variety of applications intended to be supported by the common language(s), for several sets of weightings to be determined; e.g., one set emphasizing efficiency at the expense of language power and security, another set sacrificing efficiency where necessary to gain reliability.) There is no question that the establishment of priorities for the Tinman characteristics is a complex task, but unless there is agreement on what is a requirement as opposed to a desirable facility or a luxury which could be omitted, it will be impossible for DoD to assess objectively the language evaluations performed by different contractors.

c. "State-of-the-Art" Features. Although the Tinman requires in M1 that "the language will be composed from features which are within the state of the art", some of the needed characteristics are more in the realm of language research. For example, the notion that a simple kernel coupled with a general set of extension facilities is sufficient to define a "real" programming language (implicit in A2 and E1) has been an elusive goal for researchers in extensible languages for nearly a decade. As another illustration, the issue of data abstraction (E1, E5, E8) is currently an active research area, and the implemented HOLs which provide this facility are in use in academic as opposed to commercial or governmental environments.

3. Recommendation concerning Hardware.

As part of requirement M3, the Tinman states: "The language will be defined as if it were the machine level language of an abstract digital computer." We recommend strongly that DoD follow a logical extension of this concept and undertake the establishment of common language(s) together with common hardware on which the language features can be efficiently implemented. For example, if the hardware provided array bounds checking and tagged memory, then efficiency and reliability need not be traded off in the performance of subscripting and initialization checking. It is worthwhile here to recall the comments of Dijkstra on an earlier version of the Tinman requirements [9, pp. 2, 3]:

> ...in the past, when we used "low level languages" it was considered to be the purpose of our programs to instruct our machines; now, when using "high order languages", we would like to regard it as the purpose of our machines to execute our programs. Run time inefficiency can be viewed as a mismatch between the program as stated and the machinery executing it. The difference between past and present is that in the past the programmer was always blamed for such a mismatch: he should have written a more efficient, more "cunning" program! With the programming discipline acquiring some maturity, with a better understanding of what it means to write a program so that the belief in its correctness can be justified, we tend to accept such a program as "a good program" if matching hardware is thinkable, and if with respect to a given machine [the] aforementioned mismatch then occurs, we now tend to blame that computer as ill-designed, inadequate and unsuitable for proper usage. In such a situation there are only a few true ways out of the dilemma
> 1) accept the mismatch
> 2) continue bit pushing in the old way, with all the known ill effects
> 3) reject the hardware, because it has been identified as inadequate...
> I cannot suggest strongly enough each time to select one of the three ways out of the dilemma, and not to mix them. When the second alternative

"continue bit pushing in the old way, with all the known ill effects" is chosen, let that be an activity with which the HOL project does not concern itself: if it does, the "ill effects" will propagate through the whole system.

## Section III. SUITABILITY OF CANDIDATE LANGUAGES

### 1. Introduction.

In this section we summarize the strengths and weaknesses of this report's six candidate HOLs with respect to the Tinman requirements. The approach presented here is to identify the main objectives which the HOLs attempt to fulfill (e.g., learnability, power, efficiency, reliability, maintainability, transportability) and to contrast these with the goals emphasized in the Tinman characteristics. Software reliability and maintainability are the objectives which come through most strongly in the Tinman; transportability, power, and learnability appear as secondary goals; and efficiency receives somewhat less emphasis than the others. We repeat our agreement with these priorities and point to our comments in paragraph 9.I.3 above for a method of achieving efficiency with a Tinman-like language. A summary of the evaluations is given in Table VII at the end of this chapter.

### 2. TACPOL.

a. The main attraction of TACPOL is its simplicity. The number of features available in the language is relatively small, and the rules governing the use of these features are generally straightforward. As a result, it is expected that compilers for TACPOL can be efficient. In addition, because of the nature of the facilities provided in TACPOL, efficiency in the run-time code is achievable.

b. One of the benefits of a simple language is learnability, and this goal is achieved fairly successfully through the reference manuals ([3] and [6]). These documents are understandable; [3] provides a formal definition, and [6] serves more as a user's guide. The execution model presented in [3] is particularly helpful in describing the behavior of the various language constructs.

c. Unfortunately, the attractive qualities of TACPOL come at the expense of goals which are equally if not more important for a common HOL. The major problems arise in the areas of security, portability, language expressibility and maintainability.

d. With respect to security, TACPOL has a serious gap in its type checking. For example, the overlaying of data objects of different types is built into "reference" parameter passage, which allows subtle errors to be undetectable by the compiler.

e. Portability was apparently not a goal of TACPOL. For example, the limits on the size of scalars are predicated on the existence of a particular target machine architecture. A more serious problem is that the language abounds in "undefined" conditions (e.g., the result of fixed point overflow, out of bounds subscripts) whose effects are dependent on the implementation. In addition, the absence of floating-point data and formatted I/O are major deficiencies in the language.

f. In attempting to facilitate the generation of efficient code, TACPOL has frequently made restrictions on various features. These restrictions, however, are to the detriment of language expressibility, impairing both writability and readability of programs, and thus increasing the effort involved in program maintenance. To take one example, it is not permitted to WRITE a literal value to a file; instead, this value must first be assigned to a variable. As another example, one may not pass by reference a scalar quantity obtained by subscripting an array; instead, one must pass the entire array.

g. In summary, TACPOL (in its present form) diverges too widely from Tinman requirements, and, because of the scope of modifications needed to bring the language into compliance, such an approach is not practical. Since many of the major differences result from TACPOL's basic design philosophy (not simply from features lacking in the language), it is also recommended that TACPOL not be used as the base for the design of a common language.

3. CS-4.

a. The main strengths of CS-4 are in the areas of program reliability, maintainability, transportability, and language power. Reliability and maintainability were fundamental design goals and are illustrated in such facilities as strong type checking (even across separate compilations), "information hiding" (which prevents the user of a program or data abstraction module from writing

programs which depend upon how the module operates), an exception-handling facility under user control, and protection features for data shared among concurrent processes. Transportability is illustrated most clearly in the arithmetic data types (e.g., the user is required to specify range bounds for integer variables and range and precision for reals) and in the operating system interface (which supports features such as I/O and process management in a standard fashion). The facilities cited in this paragraph also illustrate the power of the language.

b. Weaknesses in CS-4 arise in three areas: language complexity; incompleteness of specification; and needed features which are missing from CS-4. The complexity of the language is due to several factors, the most significant of which is the attempt of the language design to obtain reliability and transportability without sacrificing efficiency. As a result the language includes compiler directives to turn off run-time checking in contexts like array subscripting and union component selection; also, there is an elaborate set of rules concerning the actual precision of real and fraction data. Another cause for the language complexity is that interactions between features sometimes result in "special cases" which have to be described; e.g., the rules constraining the kinds of parameter passage allowed for user-defined implicit procedures in data abstractions. One possible way to eliminate a major cause of these interactions is to remove the data abstraction facility from the language, replacing it with a simpler type definition mechanism such as is found in PASCAL. Although data abstraction directly satisfies some of the Tinman requirements (B1, E1, E5, E8), this is currently a research area among programming language investigators, and many of the advantages of data abstraction can be realized through other CS-4 facilities (most notably, separate compilation).

c. A second weakness in CS-4, mentioned above, that of incompleteness of specification, is apparent in the Operating System Interface and also in the reference manual. It is likely that attempts at a complete specification will reveal opportunities for reducing complexity in the language. The third weakness, needed features missing from CS-4, is illustrated by the absence from the language of pointers, recursion, operator extension, and a compile-time facility such as a macro processor. (It might be noted that

the absence of pointers and recursion from CS-4 does not imply that storage allocation can be carried out completely at compile-time; the presence of run-time determinable array bounds thwarts this goal.)

d. Among the six candidate HOLs considered in this report, CS-4 comes closest to satisfying the Tinman requirements. The aspect of CS-4 which separates it most clearly from the other five HOLs is that constructs which promote sound programming methodology (e.g., type definition and type checking, "structured" control flow facilities) are enforced by the language itself. (Although it is perhaps possible to obtain some of the desirable attributes of such constructs in other languages, with a macro facility and/or a management-imposed programming discipline, it is preferable for the language to contain the enforcement mechanism.) The changes required to bring CS-4 into compliance with the Tinman are minor to moderate in scope and can be made without violating the spirit of the language; these changes are primarily the additions cited in paragraph c above. Thus, CS-4 is suitable, and we recommend its selection, as a basis on which to define a language that satisfies the Tinman requirements.

4. JOVIAL (J73/I).

a. The main strengths of J73/I are, first, that its features are realizable with a fairly high degree of efficiency; second, that its macro (DEFINE) facility makes it possible to overcome some of the language's drawbacks; and third, that it is a successor to a family of HOLs which has been widely used for embedded computer applications (at least in the Air Force).

b. Weaknesses of JOVIAL arise in connection with transportability, simplicity, reliability, readability and power. The fact that machine dependencies permeate the entire language rather than occurring in encapsulated, specialized facilities; the silence of the defining document concerning the detection of errors; and the fact that there is no specification of I/O, process management, and exception handling, combine to defeat the goal of software portability in JOVIAL. Despite its relative smallness, JOVIAL is not an especially simple language; the variety of special cases concerning such facilities as parameter passing, type matching, and data allocation illustrate some

of the complexity. Reliability was sacrificed to obtain run-time (and sometimes compile-time) efficiency; examples are the presence of storage overlaying as opposed to discriminated union, a pointer facility which is insecure, and the absence of type checking when tables or blocks are passed as parameters. Shortcomings with respect to readability are revealed most clearly in the notation for variable declarations, which makes use of a variety of non-obvious abbreviations. The absence of facilities for the definition of data types detracts seriously from the power of the language.

c. Because of the clash between the objectives of JOVIAL and those of the Tinman, JOVIAL fails to satisfy a large number of the Tinman's requirements, and it is not practical to attempt to modify the language to bring it into compliance. For a similar reason, we recommend that JOVIAL not be used as the basis for the design of a common Tinman-oriented language: many of the basic design tenets of JOVIAL would have to be modified, and the resultant language would bear little resemblance to JOVIAL.

5. FORTRAN

a. The main strengths of FORTRAN lie in the efficiency of most of its constructs, its relative simplicity, and its widespread availability (at least for previous versions of the language). In addition, the presence of character data solves one of the chief problems with earlier version of FORTRAN.

b. The language's weaknesses occur in connection with the objectives of power, portability and reliability. The absence of a mechanism for defining heterogeneous data objects (such as PL/I "structures"), the lack of a type definition facility, restriction to static allocation (and thus the absence of a pointer facility and recursion) are major deficiencies. Despite the widespread availability of FORTRAN, there are basic conflicts with portability: floating point data are specified as single or double precision (a machine dependent notion) as opposed to a range of values with an indicated minimum number of digits of precision; also, the reference document leaves to the implementation the decision as to whether a program is legitimate or not. The compromising of program reliability is illustrated in storage-oriented features such as COMMON

and EQUIVALENCE and the lack of type checking in contexts such as passage of array parameters.

c. The comments in paragraphs 2g and 4c above concerning TACPOL and JOVIAL are applicable also to FORTRAN. It is not realistic to attempt to modify FORTRAN to bring it into compliance with the Tinman; we also advise that FORTRAN not be used as the basis for the design of a common language. Programming methodology has progressed considerably in the twenty-or-so years since many of FORTRAN's features first appeared, and a common language should take advantage of these developments.

6. COBOL

a. The main strength of COBOL is that it is a language which is well-matched to its intended application area (business data processing). With its diversity of character-oriented I/O facilities, COBOL is adequate for the specification of simple, transaction-based algorithms. The hierarchical program organization required in COBOL (e.g., the four divisions) coupled with the English-like notation in the PROCEDURE DIVISION, contribute toward program readability. The availability of COBOL (it is the most widely used HOL) is an obvious strength, and we point out also that the design of hardware architectures to support efficiently the constructs of the language (see our recommendations in paragraph 9.I.3 above) has been successfully engineered (e.g., the Burroughs B3700).

b. There are a variety of fundamental weaknesses in COBOL which make it unsuitable for application outside business data processing. First, the data types and data definition facility are inadequate; particular weaknesses are in the areas of arithmetic and boolean data. Second, the control features offered by COBOL are not powerful enough to support general-purpose programming. The ability to invoke subroutines is limited (and distributed between two separate facilities), and there is no block-structure. Third, the language is fairly complex, and many of the rules are plagued by special cases and exceptions, owing to the interactions between language features. Programming practices which tend to be error-prone (e.g., overlaying data via a "free" as opposed to "discriminated" union) are often necessary in COBOL programs. Fourth, there are a variety of implementation dependencies in the language which

defeat the goal of portability. A striking example is the language definition's failure to specify floating-point data and arithmetic.

c. For many of the same reasons given in connection with TACPOL, JOVIAL, and FORTRAN, it is unrealistic to attempt to modify COBOL to bring it into compliance with the Tinman. Moreover, such factors as the language's overriding orientation toward character data and the absence of definitional facilities make COBOL a poor basis for the design of a common language.

7. PL/I

a. The main strengths of PL/I stem from the power of the language and the support provided by various manufacturers (e.g., IBM, Honeywell, and Burroughs). Significant examples of the language's expressive power are its procedure definition facility (e.g., its generic mechanism), its exception handling features, and its variety of I/O and character manipulation constructs.

b. Weaknesses in PL/I emerge in connection with the goals of learnability, efficiency, reliability, maintainability, and (despite the fact that this is also one of the language's strong points) expressive power. PL/I is a large and complex language with features sometimes interacting in obscure ways. The overhead required at run-time to support its constructs (e.g., implicit conversion) is considerable. The widespread appearance of defaults and the insecurity of the pointer facility interfere with reliability and maintainability. The absence of a type-definition mechanism (such as is found in PASCAL) is a serious deficiency and detracts from the power of the language.

c. The substantial differences in philosophy between PL/I and the Tinman imply that an attempt to modify PL/I to bring it into compliance with the Tinman's requirements would amount to a redesign of a large part of the language. This is not a realistic approach. PL/I was based on the idea that a HOL should allow its programmers the widest freedom in using the features of the language. However, recent developments in programming methodology have revealed that <u>security</u> of language constructs, and not unlimited <u>freedom</u>, is desirable in the interests of readability,

reliability, and maintainability of software. Because of these fundamental differences between PL/I and the Tinman, we recommend that PL/I not be used as the basis for the design of a common HOL.

8. Summary of Recommendations.

a. With respect to choosing a HOL either as is or as the base for the design of a common language to satisfy the Tinman requirements, CS-4 is superior to the other five candidate HOLs considered in this report (TACPOL, JOVIAL J73/I, FORTRAN, COBOL and PL/I). This conclusion results from the fact that the high-priority objectives of the Tinman (software reliability, maintainability) are met most closely by CS-4. The application of the Evaluation Tool lends credence to this result. Even if we take into account the inaccuracies to which such a quantitative approach is subject, CS-4 emerged with a significantly higher score for technical merit (approximately 76 out of 100) than the other candidate HOLs (51 for PL/I and JOVIAL, 49 for TACPOL, 47 for COBOL, and 42 for FORTRAN).

b. A second recommendation is that the design of a common hardware architecture be undertaken in conjunction with the HOL effort, to ensure an efficient match between language and machine. The design of a language as powerful as one which would comply with the Tinman will likely fail by virtue of its complexity if it is constrained to be efficiently supportable on the variety of hardware configurations currently in existence within DoD.

c. A third recommendation is that the critiques of the various candidate HOLs be used as input for the respective standards groups, insofar as these reviews suggest improvements within the spirit of the languages.

d. A fourth recommendation is that the Tinman be revised, simplified and clarified in accordance with our comments in Appendix III.

TABLE X. SUMMARY OF LANGUAGE EVALUATIONS (*)

| | TACPOL | CS-4 | JOVIAL | FORTRAN | COBOL | PL/I | |
|---|---|---|---|---|---|---|---|
| A01 | P L | T | P L | P ML | FP L | P S | A01 |
| A02 | P L | P L | P L | P L | P L | PT S | A02 |
| A03 | F L | P S | P S | F SM | F SM | P S | A03 |
| A04 | P S | P L | F L | F L | T | P ML | A04 |
| A05 | P L | P S | F L | F L | P L | P L | A05 |
| A06 | P L | PT S | P S | P SL | P L | P S | A06 |
| A07 | F L | PT M | P L | F L | P L | F L | A07 |
| B01 | P L | PT S | P L | P M | P L | P L | B01 |
| B02 | P S | P S | P L | P SM | P L | P S | B02 |
| B03 | P M | PT L | PT M | P M | P L | T | B03 |
| B04 | P L | T | PT S | PT S | PT SM | T | B04 |
| B05 | F LS | T | P L | PT SM | P S | F MS | B05 |
| B06 | P M | T | PT S | P S | P S | P SM | B06 |
| B07 | F M | P S | F L | F M | F L | P MS | B07 |
| B08 | P L | PT M | P L | P LS | F L | F S | B08 |
| B09 | P L | T | P L | P SM | PT S | PT L | B09 |
| B10 | P L | PT S | F L | P L | PT L | PT L | B10 |
| B11 | F M | T | P MS | P SM | F L | P M | B11 |
| C01 | F SL | T | F SM | F SL | P SM | P S | C01 |
| C02 | PT S | T | PT S | PT S | P S | P S | C02 |
| C03 | T | T | T | T | F L | T | C03 |
| C04 | F SL | P S | P MS | P SM | F S | P SM | C04 |
| C05 | P M | P L | F L | T | T | P M | C05 |
| C06 | P M | PT M | P L | P M | F L | P ML | C06 |
| C07 | P L | P L | P L | P SM | F L | F ML | C07 |
| C08 | P SM | P M | F L | F ML | F M | P M | C08 |
| C09 | F L | F M | F M | F ML | F L | F ML | C09 |

---

(*) Key for Degree of compliance: "T" means "Totally meets the requirement;" "P" means "Partially meets the requirement;" "F" means "Fails to meet the requirement;" "U" means "Unknown from the available documents if the requirement is satisfied."
Key for Scope of modifications: "S" means "small," "M" means "moderate," "L" means "large." If two entries appear, the first applies to the language and the second to the implementation.
Example: The entry "F SL" for TACPOL and C01 means that TACPOL fails to satisfy C01, and bringing TACPOL into compliance with the requirement has small impact on the language but a large effect on implementations.

TABLE X (continued)

| | TACPOL | CS-4 | JOVIAL | FORTRAN | COBOL | PL/I | |
|---|---|---|---|---|---|---|---|
| D01 | P M | T | P MS | PT S | F M | F SM | D01 |
| D02 | PT S | PT S | P L | P S | T | P SM | D02 |
| D03 | F ML | T | P SM | P MS | F M | P SL | D03 |
| D04 | P M | P L | P M | F L | P L | F L | D04 |
| D05 | P ML | PT L | P L | P L | P L | P SM | D05 |
| D06 | F L | F L | P L | F L | F L | P L | D06 |
| E01 | F L | P L | P L | F L | F L | P L | E01 |
| E02 | F L | P L | F L | F L | F L | F L | E02 |
| E03 | PT S | T | T | PT M | T | P S | E03 |
| E04 | F L | F L | F L | F L | F L | F L | E04 |
| E05 | F L | T | F L | F L | F L | F L | E05 |
| E06 | P M | T | P L | F L | PF L | P L | E06 |
| E07 | P M | PT M | P L | P SM | P L | P L | E07 |
| E08 | F L | T | F L | F L | F L | F L | E08 |
| F01 | P S | T | PT M | P M | FP L | PT M | F01 |
| F02 | F L | T | P S | F ML | F L | P L | F02 |
| F03 | T | T | T | P L | FP L | P S | F03 |
| F04 | P SL | P S | PT S | T | P L | PU MS | F04 |
| F05 | P SL | T | T | F S | PT SM | PU MS | F05 |
| F06 | P SL | T | T | F S | T | FU MS | F06 |
| F07 | P L | PT M | F L | P L | P L | P L | F07 |
| G01 | P L | PT SM | P L | P L | P L | P L | G01 |
| G02 | P S | PT S | P SM | P MS | PT S | F SM | G02 |
| G03 | PT S | P S | PT S | P M | P M | P SM | G03 |
| G04 | P S | PT S | P S | P M | P L | P M | G04 |
| G05 | F SL | F SM | F SM | F SL | F L | PT | G05 |
| G06 | F L | PT S | F L | F L | F L | F L | G06 |
| G07 | P L | PT S | F L | F L | P L | P L | G07 |
| G08 | P L | P M | F L | F L | F L | F L | G08 |
| H01 | PT S | PT S | P MS | P L | P L | P M | H01 |
| H02 | T | T | T | T | T | T | H02 |
| H03 | T | P S | T | T | T | P S | H03 |
| H04 | PT S | PT S | P S | PT S | P S | P S | H04 |
| H05 | PU S | P S | P S | F S | P S | P S | H05 |
| H06 | P S | P S | P S | P S | P LS | F S | H06 |
| H07 | P S | P S | PT S | P S | P S | PT S | H07 |
| H08 | T | T | T | T | T | F S | H08 |
| H09 | P L | P L | P MS | P L | F L | P LM | H09 |
| H10 | P S | T | P MS | T | P S | P S | H10 |

TABLE X (continued)

| | TACPOL | CS-4 | JOVIAL | FORTRAN | COBOL | PL/I | |
|---|---|---|---|---|---|---|---|
| I01 | F L | P M | F LM | F L | F ML | F L | I01 |
| I02 | P M | P SM | P SM | F L | P L | P SM | I02 |
| I03 | F M | PT S | T | F SM | F S | F S | I03 |
| I04 | F M | P L | T | F SM | F M | F MS | I04 |
| I05 | PT S | P M | P S | PT S | F L | F LM | I05 |
| I06 | P S | P S | P S | P S | P S | F S | I06 |
| I07 | P M | T | PT S | T | T | T | I07 |
| J01 | PT S | PT M | PT L | PT S | P M | F L | J01 |
| J02 | FU SM | TU | F SL | F SL | TU | F S | J02 |
| J03 | P M | PT L | P SM | P S | P M | F L | J03 |
| J04 | P ML | PT SL | P L | F SL | P L | P L | J04 |
| J05 | F S | T | F M | F S | F L | F SM | J05 |
| K01 | P L | T | T | F SL | F L | PU | K01 |
| K02 | U | P S | T | U S | P L | PU ML | K02 |
| K03 | U L | U | U | U | PU | U | K03 |
| K04 | U SL | U | U | U | U L | U | K04 |
| K05 | F L | FU M | F S | F S | F L | PU | K05 |
| L01 | U | U | F LS | F SL | F SL | F S | L01 |
| L02 | U | U | U | F SL | F L | T | L02 |
| L03 | U S | P L | U | T | U L | FU | L03 |
| L04 | U L | U | U | U S | U | U | L04 |
| L05 | U | U | U | U | U | U | L05 |
| L06 | U | U | U | F SM | U | U | L06 |
| L07 | U | U | F L | F SM | U | U | L07 |
| L08 | U | U | T | T | U | T | L08 |
| L09 | U M | U | T | U | U | U | L09 |
| M01 | P L | P M | T | T | P L | P M | M01 |
| M02 | P S | P M | F L | P S | P L | P M | M02 |
| M03 | U | U | F L | P L | U | U | M03 |
| M04 | U | U | U | U | U | U | M04 |
| M05 | U | U | U | U | U | U | M05 |
| M06 | U | U | U | U | U | U | M06 |

# LITERATURE CITED

## REPORTS


1. American National Standards Committee X3J3. Draft Proposed ANS FORTRAN, published in SIGPLAN Notices, Volume 11, Number 3, March 1976.

2. American National Standards Technical Committee X3J1 and European Computer Manufacturers Association Technical Committee TC10. Draft Proposed American National Standard Programming Language PL/I: BASIS/1-12, Document BSR X3.53; February 1975; and Errata for BSR X3.53, Document X3J1/399; January 1976.

3. Litton Systems, Inc., Data Systems Division. CPCEI Specification for Compiler/Assembler for Fire Direction System Artillery AN/GSG-10() (V). Specification EL-CG-00043082C; Document 595909-600C; Contract DAAB07-68-0154; April 1971.

4. Brosgol, Benjamin M., Timothy A. Dreisbach, James L. Felty, Joel R. Lexier, and Gary M. Palter. CS-4 Language Reference Manual; Document IR-130-2; Contract N00123-74-C-0634; Intermetrics, Inc.; October 1975.

5. Grimes, Donald E., and Joel R. Lexier. CS-4 Operating System Interface; Document IR-130-2; Contract N00123-74-C-0634; Intermetrics, Inc.; October 1975.

6. Litton Data Systems, Inc., TACPOL Reference Manual Programming Support System. Document USACSCS-TF-4-1; Contract DAAB07-68-C-0154; June 1975.

7. Brosgol, Benjamin M., and James L. Felty. Language Requirements Report; Document IR-179-2; Contract DAHC26-76-C-0006; Intermetrics, Inc.; January 1977.

8. American National Standards Institute. American National Standard Programming Language COBOL, X3.23-1974.

9. Dijkstra, E.W. "On a language proposal for the Department of Defense," EWD514, September 17, 1975.

10. Robinson, L., M.W. Green, R. E. Shostak, and J. M. Spitzen. The Verification of COBOL Programs. Contract DAHC-04-75-0011; Stanford Research Institute; March 1976.

JOURNAL/MAGAZINE ARTICLES

11. Rothenbuecher, O. H. "The Top 50 Companies in the Data Processing Industry." Datamation, 22 (June 1976) pp. 48-59.

12. Knuth, Donald E. "Structured Programming with Go To Statements." ACM Computing Surveys, 6 (December 1974) pp. 261-301.

13. Reifer, D. J. "The Structured FORTRAN Dilemma." SIGPLAN Notices, 11 (February 1976) pp. 30-32.

GOVERNMENT PUBLICATIONS

14. Air Force Systems Command, Rome Air Development Center. JOVIAL J73/I Specification. July 1976.

15. Marcotty, Michael. "Suitability of PL/I as the Army Programming Language," Centacs Report No. 47, U.S. Army Electronics Command, Fort Monmouth, N.J. June 1975.

16. Department of Defense Requirements for High Order Computer Programming Languages "Tinman"; June 1976.

BOOKS

17. Sammet, Jean E. Programming Languages: History and Fundamentals. Englewood Cliffs, N.J., Prentice-Hall, 1969.

## Appendix I

## HIGH ORDER LANGUAGE AVAILABILITY QUESTIONNAIRE

Please list each High Order Language being used in your project. For each language, please answer the questions in Parts I and II.

Part I. General Information.

A. What are the main reasons that this HOL was chosen? If there were any alternate languages under consideration, what are the main reasons that none of these was selected?

B. What are the host and target computers for the compiler?

C. Is more than one dialect of the language being used? If so, why?

D. How many programmers are using the language? (Count programmers using the language part time by estimating the fraction of time spent.)

Part II. Availability Data.

This part consists of three sections. In Section A, you are asked to provide scores for various factors comprising the components of the language availability measure. In Section B, you are asked to supply, for each component, relative weightings for the factors constituting that component. In Section C, you are asked to supply relative weightings for the components themselves.

A. Scores for factors of language-availability components.

Three components (compiler availability, quality of language-oriented support tools, quality of documentation) are broken down into factors. For each factor, you are asked to supply a score (between 0 and 10) reflecting the way in which that factor is met at your site.

A fourth component ("other") is listed to account for factors which might not be conveniently included in the other three. If you wish to supply information in this category, please attach a supplementary sheet explaining the factors

you are scoring. In addition, the compiler availability, tools, and documentation components include a factor named "other" so that additional factors in these categories can be entered.

1. Compiler availability.

a. Compile-time diagnostics.

In the score for this facility, take the following issues into account: Are warnings produced for probable errors (such as code which can never be executed)? Does the compiler recover from program errors without introducing bogus ones? Are error messages specific and meaningful? If the language reference manual defines a set of compile-time errors, are all of these detected? Is compile-time data-type checking performed?

Score for compile-time diagnostics:

b. Run-time diagnostics.

Issues to take into account in estimating the score here are: subscript bounds checking, checking for null pointers on dereferencing operations, meaningful error messages, debug facilities (such as statement or procedure tracing, symbolic dumps, breakpoints).

Score for run-time diagnostics:

c. Object code efficiency.

The score here should be relatively high if the compiled code is suitably efficient for the application area. Inversely, if either assembly language must be used or useful features in the language must be avoided because of object code inefficiency, a low score is appropriate.

Score for object code efficiency:

d. Compiler reliability.

A low score here should reflect a situation in which the compiler has contained serious errors, requiring a long time to correct, resulting in delays or other problems for project personnel. If compiler errors have proved to be relatively infrequent and easy to correct, then the score for this goal should be high.

Score for compiler reliability:

e. Compiler smallness and speed

A low score here should reflect a large, slow compiler which has been a source of bottlenecks in the project. A compact, fast compiler should receive a high score for this goal.

Score for compiler smallness and speed:

f. Listings produced by compiler

Take into account here such attributes as: automatic formatting and indenting, ease of finding errors, cross-references, options for listing the object code.

Score for listings produced by compiler:

g. Other

Explain on attached page. Score:

2. Quality of language-oriented support tools

For each of the following tools which is available, evaluate its performance on a 0 to 10 scale (if the tool is not available, rate it as 0).

a. Language-specific text-editor. Score:

b. Verification aids (such as test program generators, static flow analyzer). Score:

c. Program instrumentation aids (such as dynamic statistics-gathering packages). Score:

d. Other (explain on attached sheet). Score:

3. Quality of language documentation

Rate each of the following on a 0 to 10 scale.

a. Language definition document

A high score here should reflect a document which is complete, unambiguous, and readable by an experienced programmer. Deduct points if there have been questions regarding program legality which could not be answered using the document.

Score for language definition document:

b. Programmer's guide

This document is a manual which combines a description of the language (less formal than the definition document)

with a specification of implementation dependencies (e.g., operating-system-oriented facilities, maximum and minimum numeric values). As the main "working document" of the programmer, this manual should be organized to facilitate quick reference but should not sacrifice readability.

Score for programmer's guide:

c. Primer

The primer (possibly a textbook) is an introduction to the language for the novice programmer. Readability is the main goal; the document should contain examples for each feature explained and should also include a set of exercises for the reader.

Score for primer:

d. Other (explain on attached sheet). Score:

4. Other factors related to language availability (explain on attached sheet). Score:

B. Weightings for factors of language-availability composition.

The scores for the various components in Section A above will be weighted by the values which you supply here.

1. Compiler availability

Assign a relative weight to each of the factors associated with the compiler availability. This set of weights should reflect the priorities particular to your project. Each weight should be a fraction (between 0.0 and 1.0), and the sum of the individual weights for the compiler availability factors should be 1.0.

| | | |
|---|---|---|
| a. | Compile-time diagnostics | Weight |
| b. | Run-time diagnostics | Weight |
| c. | Object code efficiency | Weight |
| d. | Compiler reliability | Weight |
| e. | Compiler smallness and speed | Weight |
| f. | Listings produced by compiler | Weight |
| g. | Other | Weight |

Sum = 1.0

2. Quality of language-oriented support tools.

Assign a relative weight to each of the factors associated with language-oriented support tools. This set of weights should reflect the priorities particular to your project. Each weight should be a fraction (between 0.0 and 1.0), and the sum of the individual weights for the support tool factors should be 1.0.

| | | |
|---|---|---|
| a. | Language-specific text editor | Weight |
| b. | Verification aids | Weight |
| c. | Program instrumentation aids | Weight |
| d. | Other | Weight |
| | Sum = | 1.0 |

3. Quality of language documentation.

Assign a relative weight to each of the factors associated with language documentation. This set of weights should reflect the priorities particular to your project. Each weight should be a fraction (between 0.0 and 1.0), and the sum of the individual weights for the language documentation factors should be 1.0.

| | | |
|---|---|---|
| a. | Language definition document | Weight |
| b. | Programmer's guide | Weight |
| c. | Primer | Weight |
| d. | Other | Weight |
| | Sum = | 1.0 |

4. Other

If you supplied a set of factors with individual scores, assign relative weights to these factors using the same conventions as for components 1 through 3 above.

C. Weightings for the language availability components

The scores supplied in section A and the weightings given in section B will determine a set of four intermediate scores, each between 0 and 10, one for each component (compiler, tools, documentation, other). In the final step in the procedure for deriving a single score for language availability,

assign weights to each of these components. Again, base these weights on the priorities specific to your project. These weights should be between 0.0 and 1.0 and sum to 1.0.

| | | |
|---|---|---|
| 1. | Compiler availability | Weight |
| 2. | Quality of language-oriented support tools | Weight |
| 3. | Quality of language documentation | Weight |
| 4. | Other | Weight |
| | Sum = | 1.0 |

Appendix II

# DEPARTMENT OF DEFENSE REQUIREMENTS FOR HIGH ORDER COMPUTER PROGRAMMING LANGUAGES "TINMAN"

June 1976

Section IV

A. DATA AND TYPES
1. Typed Language
2. Data Types
3. Precision
4. Fixed Point Numbers
5. Character Data
6. Arrays
7. Records

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable, and component of composite data structures be explicitly specified in the source programs.

By the type of a data object is meant the set of objects themselves, the essential properties of those objects and the set of operations which give access to and take advantage of those properties. The author of any correct program in any programming language must, of course, know the types of all data and variables used in his programs. If the program is to be maintainable, modifiable and comprehensible by someone other than its author, the the types of variables, operations, and expressions should be easily determined from the source program. Type specifications in programs provide the redundancy necessary to verify automatically that the programmer has adhered to his own type conventions. Static type definitions also provide information at compile time necessary for production of efficient object code. Compile time determination of types does not preclude the inclusion of language structures for dynamic discrimination among alternative record formats (see A7) or among components of a union type (see E6). Where the subtype or record structure cannot be determined until run time, it should still be fully discriminated in the program text so that all the type checks can be completed at compile time.

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

These are the common data types and type generators of most programming languages and object machines. They are sufficient, when used with a data

definition facility (see E6, D6, and J1), to efficiently mechanize other desired types such as complex or vector.

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

This is a specification of what the program needs, not what the hardware provides. Machine independence, in the use of approximate value numbers (usually with floating point representation), can be achieved only if the user can place constraints on the translator and object machine without forcing a specific mechanization of the arithmetic. Precision specifications, as the minimum supported by the object code, provide all the power and guarantees needed by the programmer without unnecessarily constraining the object machine realization. Precision specifications will not change the type of reals nor the set of applicable operations. Precison specifications apply to arithmetic operations as well as to the data and therefore should be specified once for a designated scope. This permits different precisions to be used in different parts of a program. Specification of the precision will also contribute to the legibility and implementability of programs.

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.

Scaled integers are useful approximations to real numbers when dealing with exact quantity fractional values, when the object machine does not have floating point hardware, and when greater precision is required than is available with the floating point hardware. Integers will also be treated as exact quantities with a step size equal to one.

A5. Character sets will be treated as any other enumeration type.

Like any other data type defined by enumeration (see E6), it should be possible to specify the program literal and order of characters. These properties of the character set would be unalterable at run time. The definition of a character set should reflect on the manner it is used within a program and not necessarily on the

print representation a particular physical device associates with a bit pattern at run time. In general, unless all devices use the same character code, run-time translation between character sets will be required. Widely used character sets, such as, USASCII and EBCDIC will be available in a standard library. Note that access to a linear array filled with the characters of an alphabet, A, and indexed by an alphabet, B, will convert strings of characters from B to A.

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

This is general enough to permit both arrays which can be allocated at compile or load time and arrays which can be allocated at scope entry, but does not permit dynamic change to the size of constructed arrays. It is sufficient to permit allocation of space pools which the user can manage for allocation of more complex data structures including dynamic arrays. The range of subscript values for any given dimension will be a contiguous subsequence of values from an enumeration type (including integers). The preferable lower bound on the subscript range will be the initial element of an enumeration type or zero, because it often contributes to program efficiency and clarity.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

This provides all that is safe to use in CMS-2 and JOVIAL OVERLAY and in FORTRAN EQUIVALENCE. It permits hierarchically structured data of heterogeneous type, permits records to have alternative structures as long as each structure is fixed at compile time and the choice is fully discriminated at run time, but it does not permit arbitrary references to memory nor the dropping of type checking when handling overlayed structures. The discrimination condition will not be restricted to a field of the record but should be any expression.

B. OPERATIONS
1. Assignment and Reference
2. Equivalence
3. Relationals
4. Arithmetic Operations
5. Truncation and Rounding
6. Boolean Operations
7. Scalar Operations
8. Type Conversion
9. Changes in Numeric Representation
10. I/O Operations
11. Power Set Operations

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

The user will be able to declare variables for all data types. Variables are useful only when there are corresponding access and assignment operations. The user will be permitted to define assignment and access operations as part of encapsulated type definitions (see E5). Otherwise, they will be automatically defined for types which do not manage the storage for their data. (See D6 for further discussion).

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

Equivalence is an essential universal operation which should not be subject to restriction on its use. There are many useful equivalence operations for some types and a language definition cannot foresee all these for user defined types. Equivalence meaning logical identity and not bit-by-bit comparison on the internal data representation, however, is required for all data types. Proper semantic interpretation of identity requires that equality and identity be the same for atomic data (i.e., numbers, characters, Boolean values, and types defined by enumeration) and that elements of a disjoint types never be identical. Consequently, its usefulness at run time is restricted to data of the same type or to types with nonempty intersections. For floating point numbers identity will be defined as the same within the specified (minimum) precision.

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration.

Numbers and types defined by enumeration have an obvious ordering which should be available through relational operations. All six relational operations will be included. It will be possible to inhibit ordering definitions when unordered sets are intended.

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

These are the most widely used numeric operations and are available as hardware operations in most machines. Floating point operations will be precise to at least the specified precision.

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

These requirements seem obvious, particularly for floating point numbers and yet many of our existing languages truncate the most significant mantissa digits in some mixed and floating point operations.

B6. The built-in Boolean operations will include "and," "or," "not," and "nor." The operations "and" and "or" on scalars will be evaluated in short circuit mode.

Short circuit mode as used here is a semantic rather than an implementation distinction and means that "and" and "or" are in fact control operations which do not evaluate side effects of their second argument if the value of the first argument is "false" or "true," respectively. Short circuit evaluation has no disadvantages over the corresponding computational operations, sometimes produces faster executing code in languages where the user can rely on the short circuit execution, and improves the clarity and maintainability of programs by permitting expressions such

as, "i<=7 & A[i] >x" which could be erroneous were short circuit execution not intended. Note that the equivalence and nonequivalence operations (see B2) are the same as logical equivalence and exclusive-or respectively.

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

Conformability will require exactly the same number of components (although a scalar can be considered compatible with any array) and one for one compatibility in type. Correspondence will be by position in similarly shaped arrays. In many situations component by component operations are done on array elements. In fact, a primary reason for having arrays is to permit large numbers of similarly treated objects to have a uniform notation. Operations on data aggregates available directly in the source language hide the details of the sequencing and thereby, simplify the program and make more optimizations available. In addition, they permit simultaneous execution on machines with parallel processing hardware. Although component by component operations will be available for built-in composite data structures which are used to define application-oriented structures, that capability will not be automatically inherited by defined data structures. A matrix might be defined using an array, but it will not inherit the array operations automatically. Multiplication for matrices would, for example, be unnatural, confusing and inconvenient if the product operator for matrices were interpreted as a component by component operation instead of cross product of corresponding row and column vectors. Component by component operations also allow operations on character strings represented as vectors of characters and allow efficient Boolean vector operations.

Transfers between arrays or records of identical logical structure are necessary to permit efficient run time conversion from one object representation to another, as might be done when data is packed to reduce peripheral storage requirements and I/O transfer times but need to be unpacked locally to minimize processing costs.

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversions operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

Implicit type conversions which represent changes in the value of data items without an explicit indicator in the program, are not only error prone but can result in unexpected run time overhead.

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

Because ranges do not form closed systems, range validation is not possible at compile time (e.g., "I:=I+1" may be a range error). At best, the compiler might point out likely range errors. (This requirement is optional for hardware installations which do not have overflow detection).

B10. The base language will provide operations allowing programs to interact with files, channels or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

Whether the referenced "files" are real or virtual and whether they are hardware devices, I/O channels or logical files depends on the object machine configuration and on the details of its operating system if present. But in any programming system I/O operations ultimately reduce to sending or receiving data and/or control information to a file or to a device controller. These can be made accessible in a HOL in an abstract form through a small set of generic I/O operations (like "read" and "write," with appropriate device and exception parameters). Note that devices and files are similar in many respects to types, so additional language features may not be required to satisfy this requirement. This requirement, in conjunction with requirement E1, permits user definition of unique equipment and its associated I/O operations as data types within the syntactic and semantic framework provided by the generic operations.

B11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

As with any data type, power sets will be useful only if there are operations which can create, select and interrogate them. Note that this provides only a very

special class of sets but one which is very useful for computations on sets of indicators, flags, and similar devices in monitoring and control applications. More general sets if desired, must be defined using the type definition facilities.

C. EXPRESSIONS AND PARAMETERS

1. Side Effects
2. Operand Structure
3. Expressions Permitted
4. Constant Expressions
5. Consistent Parameter Rules
6. Type Agreement in Parameters
7. Formal Parameter Kinds
8. Formal Parameter Specifications
9. Variable Numbers of Parameters

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

This is a semantic restriction on the evaluation order of arguments to expressions. It provides an explicit rule (i.e., left-to-right) for order of argument evaluation, but allows the implementations to alter the actual order in any way which does not change the effect. This provides the user with a simple rule for determining the effects of interactions among argument evaluations without imposing a strict rule on compilers which are sophisticated enough to detect potential side-effects and optimize through reordering of arguments when the evaluation order does not affect the result. Control operations (e.g., conditional and iterative control structures), of course, must be exceptions to this general rule since control operations are in fact those operations which specify the sequencing and evaluation rules for their arguments.

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

Care must be taken to ensure that the operator/operand structure of expressions is not psychologically ambiguous (i.e., to guarantee that the parse implemented by the language is the same as intended by the programmer and understood by those reading the program). This kind of problem can be minimized by having few precedence levels and parsing rules by allowing explicit parentheses to specify the intended execution order, and by requiring explicit parentheses when the execution order is of significance to the result within the same precedence level (e.g., "X divided by Y divided by Z" and "X divided by Y multiplied by Z"). The user will not be able to define new operator precedence rules nor change the precedence of existing operators.

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

This is an example of not imposing arbitrary restrictions and special case rules on the user of the source language. Special mention is made here only because so many languages do restrict the form of expressions. FORTRAN, for example, has a list of seven different syntactic forms for subscript expressions, instead of allowing all forms of arithmetic expressions.

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.

The ability to write constant expressions in programs has proven valuable in languages with this capability, particularly with regard to program readability and in avoiding programmer error in externally evaluating and transcribing constant expressions. They are most often used in declarations. There is no need, however, that constant expressions impose run time costs for their evaluation. They can be evaluated once at compile time or if this is inconvenient because of incompatibilities between the host and object machines, the compiler can generate code for their evaluation at load time. In any case, the resulting value should be the same (at least within the stated precision) regardless of the object machine (see D2). Allowing constant expressions in place of constants can improve the clarity, correctness and maintainability of programs and does not impose any run time costs.

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types for exception handing, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contributes to ease of learning,

implementing and using a language; allows the user to concentrate on the programming task instead of the language; and leads to more readable, understandable, and predictable programs.

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

Type transfers hidden in procedure calls with incompatible formal and actual parameters whether intentional or accidental have long been a source of program errors and of programs which are difficult to maintain. On the other hand, there is no reason why the subscript ranges for arrays cannot be passed as part of the arguments. Some notations permit such parameters to be implicit on the call side. Formal parameters of a union type will be considered conformable to actual parameters of any of the component types.

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

The first class of data parameter acts as a constant within the procedure body and cannot be assigned to nor changed during the procedures execution; its corresponding actual parameter may be any legal expression of the desired type and will be evaluated once at the time of call. The second class of data parameter renames the actual parameter which must be a variable, the address of the actual parameter variable will be determined by (or at) the time of call and unalterable during execution of the procedure, and assignment (or reference) to the formal parameter name will assign (or access) the variable which is the actual parameter. These are the only two widely used parameter passing mechanisms for data and the many alternatives (at least 10 have been suggested) add complexity and cost to a language without sufficiently increasing the clarity or power. A language with exception handling capability must have a way to pass control and related data through procedure call interfaces. Exception handling control parameters will be specified on the call side only when needed. Actual procedure parameters will be restricted to those of similar (explicit or implicit) specification parts.

C8. Specification of the type, range, precision, dimension, scale and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.

Optional formal parameter specification permits the writing of generic procedures which are instantiated at compile time by the characteristics of their actual parameters. It eliminates the need for compile time "type" parameters. This generic procedure capability, for example, allows the definition of stacks and queues and their associated operations on data of any given type without knowing the data type when the operations are defined.

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

There are many useful purposes for procedures with variable numbers of arguments. These include intrinsic functions such as "print," generalizations of operations which are both commutative and associative such as "max" and "min," and repetitive application of the same binary operation such as the Lisp "list" operation. The use of variable number of argument operations need not and will not cause relaxation of any compile time checks, require use of multiple entry procedures allow the number of actual parameters to vary at run time, nor require special calling mechanisms. If the parameters which can vary are limited to a program specified type treated as any other argument on the call side and as elements of an array within the procedure definition, full type checking can be done at compile time. There will be not prohibition on writing a special case of a procedure for a particular number of arguments.

## D. VARIABLES, LITERALS AND CONSTANTS

1. Constant Value Identifiers
2. Numeric Literals
3. Initial Values of Variables
4. Numeric Range and Step Size
5. Variable Types
6. Pointer Variables

D1. The user will have the ability to associate constant values of any type with identifiers.

The use of identifiers to represent constant values has often made programs more readable, more easily modifiable and less prone to error when the value of a constant is changed. Associating constant values with an identifier is preferable to assigning the value to a variable because it is then clearly marked in the program as a constant, can be automatically checked for unintentional changes, and often can have a more efficient object representation.

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).

Literals are needed for all atomic data types and should be provided as part of the language definition for built-in types. Regardless of the source of the data and regardless of the object machine the value of constants should be the same. For integers it should be exact and for reals it should be the same within the specified precision. Compiler writers, however, would disagree. They object to this requirement on two grounds: that it is too costly if the host and object machines are different and that it is unnecessary if they are the same. In fact, all costs are at compile time and must be insignificant compared to the life time costs resulting from object cope containing the wrong constant values. As for being unnecessary, there have been all too many cases of different values from program and data literals on the same machine because the compile time and run time conversion packages were different and imprecise.

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

The ability to initialize variables at the time of their allocation will contribute to program clarity, but a requirement to do so would be an arbitrary and sometimes costly decision to the user. Default initial values on the other hand, contribute to neither program clarity nor correctness and can be even more costly at run time. It is usually a programming error if a variable is accessed before it is initialized. It is desirable that the translator give a warning when a path between the declaration and use of a variable omits initialization. Whether a variable will be assigned is in general an unsolvable problem, but it is sometimes determinable whether assignments occur on potential paths. In the case of arrays, it it possible at compile time only to determine that some components (but not necessarily which) have been initialized. There will be provision (at user option) for run time testing for initialization.

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

Range specifications are a special form of assertion. They aid in understanding and determining the correctness of programs. They can also be used as additional information by the compiler in deciding what storage and allocation to use (e.g., half words might be more efficient for integers in the range 0 to 1000). Range specifications also offer the opportunity for the translator to insert range tests automatically for run time or debug time validation of the program logic. With the ranges of variables specified in the program, it becomes possible to perform many subscript bounds checks at compile time. These bounds, checks, however, can be only as valid as the range specifications which cannot in general be validated at compile time. Range specifications on approximate valued variables (usually with floating point implemetation) also offer the possibility of their implementation using fixed point hardware.

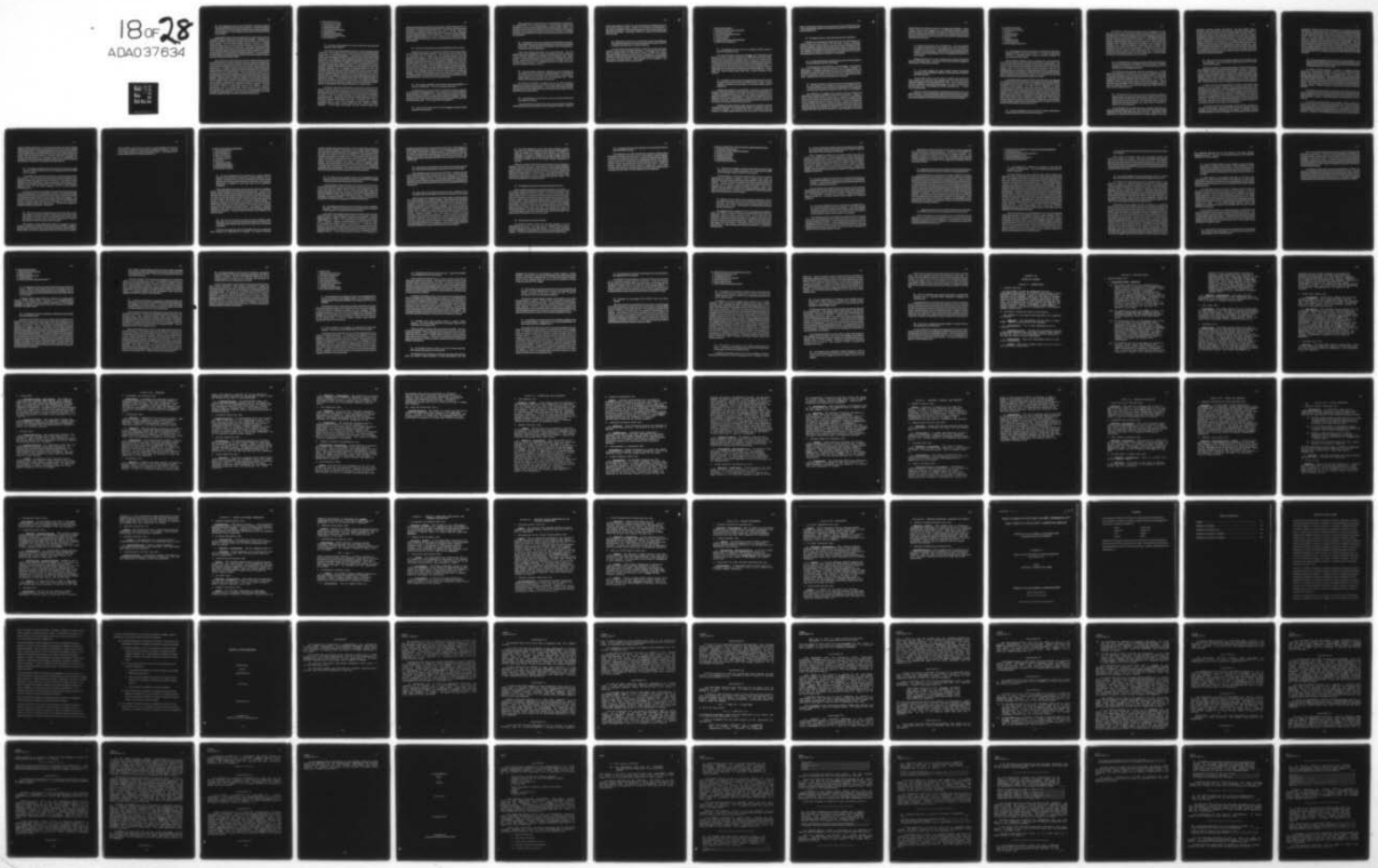
D5. The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.

There should not be any arbitrary restrictions on the structure of data. This permits arrays to be components of records or arrays and permits records to be components of arrays.

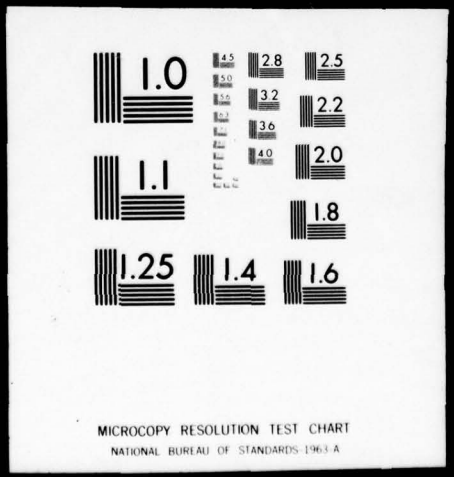
AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

18 OF 28
ADA037634



18 OF 28
ADA037634



D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

Assignment to a pointer variable will mean that the variable's name is to act as an additional label (or reference) on the datum being assigned. Assignment to a nonpointer variable will mean that the variable's name it to label a copy of the object being assigned. For data without alterable component structure or alterable component values, there is no functional difference between reference to multiple copies and multiple references to a single copy. Consequently, pointer/nonpointer will be a property only of variables for composite types and of composite array and record components Because the pointer/nonpointer property applies to all variables of a given type, it will be specified as part of the type definition. The use of pointers will be kept safe by prohibiting pointers to data structures whose allocation scope is narrower than that of the pointer variable.

Such a restriction is easily enforced at compile time using hierarchical scope rules providing there is no way to dynamically create new instances of the data type. In the latter case, the dynamically created data can be allocated with full safety using a (user or library defined) space pool which is either local (i.e., own) or global to the type definition. If variables of a type do not have the pointer property then dynamic storage allocation would be required for assignment unless their size is constant and known at the time of variable allocation. Thus, the nonpointer property will be permitted only for types (a) whose data have a structure and size which is constant in the type definition or (b) which manage the storage for their data as part of the type definition. Because pointers are often less expensive at run time than nonpointers and are subject to fewer restrictions, the specification of the nonpointer property will be explicit in programs (this is similar to the Algol-60 issue concerning the explicit specification of "value" (i.e., nonpointer) and "name" (i.e. pointer). The need for pointers is obvious in building data structures with shared or recursive substructures; such as, directed graphs, stacks, queues, and list structures. Providing pointers as absolute address types, however, produces gaps in the type checking and scope mechanisms. Type and access restricted pointers will provide the power of general pointers without their undesirable characteristics.

E. DEFINITION FACILITIES
1. User Definitions Possible
2. Consistent Use of Types
3. No Default Declarations
4. Can Extend Existing Operators
5. Type Definitions
6. Data Defining Mechanisms
7. No Free Union or Subset Types
8. Type Initialization

E1. The user of the language will be able to define new data types and operations within programs.

The number of specialized capabilities needed for a common language is large and diverse. In many cases, there is no consensus as to the form these capabilities should take in a programming language. The operational requirements dictating specific specialized language capabilities are volatile and future needs cannot always be foreseen. No language can make available all the features useful to the broad spectrum of military applications, anticipate future applications and requirements or even provide a universally "best" capability in support of a single application area. A common language needs capability for growth. It should contain all the power necessary to satisfy all the applications and the ability to specialize that power to the particular application task. A language with defining facilities for data and operations often makes it possible to add new application-oriented structures and to use new programming techniques and mechanisms through descriptions written entirely within the language. Definitions will have the appearance and costs of features which are built into the language while actually being catalogued accessible application packages. The operation definition facility will include the ability to define new infix operators (but see H2 for restrictions). No programming language can be all things to all people, but a language with data and operation definition facilities can be adapted to meet changing requirements in a variety of areas.

The ability to define data and operations is well within the state of the art. Operation definition facilities in the form of subroutines have been available in all general purpose programming languages since at least the time of early FORTRANs. Data definition facilities have been available in a variety of programming languages for almost 10 years and reached their peak with a large number of extensible languages(Stephen A. Schuman (Ed.) Proceedings of the International Symposium on Extensible Languages, SIGPLAN Notices, Vol. 6, No. 12, December 1971. Also, C. Christensen and C.J. Shaw (Ed.), Proceedings of the Extensible Language Symposium, SIGPLAN Notices 4, August 1969.) (over 30) in 1968 and shortly thereafter. A trend toward more abstract and less machine-oriented data

specification mechanisms has appeared more recently in PASCAL(Niklaus Wirth, "An Assessment of the Programming Language PASCAL, "Proceedings of the International Conference on Reliable Software 21-23 April 1973, p. 23-30). Data type definitions, with operations and data defined together, are used in several languages including SIMULA-67(Jacob Palme, "SIMULA as a Tool for Extensible Program Products, "SIGPLAN Notices, Vc!. 9, No. 4, February 1974). On the other hand, there is currently much ferment as to what is the proper function and form of data type definitions.

E2. The "use" of defined types will be indistinguishable from built-in types.

Whether a type is built-in or defined within the base will not be determinable from its syntactic and semantic properties. There will be no ad hoc special cases nor inconsistent rules to interfere with and complicate learning, using and implementing the language. If built-in features and user defined data structures and operations are treated in the same way throughout the language so that the base language, standard application libraries and application programs are treated in a uniform manner by the user and by the translator, then these distinctions will grow dim to everyone's advantage. When the language contains all the essential power, when few can tell the difference between the base language and library definitions, and when the introduction of new data types and routines does not impact the compiler and the language standards, then there is little incentive to proliferate languages. Similarly, if typed definitions are processed entirely at compile time and the language allows full program specification of the internal representation, there need be no penalty in run time efficiency for using defined types.

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

As programmers, we should not expect the translator to write our programs for us (at least in the immediate future). If we somehow know that the translator's default convention is compatible with our needs for the case at hand we should still document the choice so others can understand and maintain our programs. Neither should we be able to delay definitions (possibly forget them) until they cause trouble in the operational system. This is a special case of requirement I1.

E4. The user will be able, within the source language, to extend existing operators to new data types.

When an operation is an abstraction of an existing operation for a new type or is a generalization of an existing operation, it is inconvenient, confusing and misleading to use any but the existing operator symbol or function named. The translator will not assume that commutativity of bulit-in operations is preserved by extensions, and any assumptions about the associativity of built-in or extended operations will be ignored by the translator when explicit parentheses are provided in an expression.

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

Types define abstract data objects with special properties. The data objects are given a representation in terms of existing data structures, but they are of little value until operations are available to take advantage of their special properties. When one obtains access to a type, he needs its operations as well as its data. Numeric data is needed in many applications but is of little value to any without arithmetic operations. The definable operations will include constructors, selectors, predicates, and type conversions.

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

The above list comprises a currently known set of useful definitional mechanisms for data types which do not require run time support, as do garbage collection and dynamic storage allocation. In conjunction with pointers (see D6), they provide many of the mechanisms necessary to define recursive data structures and efficient sparse data structures.

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.

Free union adds no new power not provided by discriminated union, but does require giving up the security of types in return for programmer freedom. Range and

subset specifications on variables are useful documentation and debugging aids, but will not be construed as types. Subsets do not introduce new properties or operations not available to the superset and often do not form a closed system under the superset operations. Unlike types, membership in subsets can be determined only at run time.

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

It is often necessary to do bookkeeping or to take other special action when variables of a given type are allocated or deallocated. The language will not limit the class of definable types by withholding the ability to define those actions. Initialization might take place once when the type is allocated (i.e., in its allocation scope) and would be used to set up the procedures and initialize the variables which are local to the type definition. These operations will be definable in the encapsulation housing the rest of the type definition.

F. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed
2. Limiting Access Scope
3. Compile Time Scope Determination
4. Libraries Available
5. Library Contents
6. Libraries and Compools Indistinguishable
7. Standard Library Definitions

F1. The language will allow the user to distinguish between scope of allocation and scope of access.

The scope of allocation or lifetime of a program structure is that region of the program for which the object representation of the structure should be present. The allocation scope defines the program scope for which own variables of the structure must be maintained and identifies the time for initialization of the structure. The access scope defines the regions of the program in which the allocated structure is accessible to the program and will never be wider than the allocation scope. In some cases the user may desire that each use of a defined program structure be independent (i.e., the allocation and accessing scopes would be identical). In other cases, the various accessing scopes might share a common allocation of the structure.

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

Limited access specified in a type definition is necessary to guarantee that changes to data representations and to management routines which purportedly do not affect the calling programs are in fact safe. By rigorously controlling the set of operations applicable to a defined type, the type definition guarantees that no external use of the type can accidentally or intentionally use hidden nonessential properties of the type. Renaming separately defined programming components is necessary to avoid naming conflicts when they are used.

Limited access on the call side provides a high degree of safety and eliminates nonessential naming conflicts without limiting the degree of accessibility which can be built into programs. The alternative notion, that all declarations which are external to a program segment should have the same scope, is inconvenient and

costly in creating large systems which are composed from many subsystems because it forces global access scopes and the attendant naming conflicts on subsystems not using the defined items.

F3. The scope of identifiers will be wholly determined at compile time.

Identifiers will be declared at the beginning of their scope and multiple use of variable names will not be allowed in the same scope. Except as otherwise explicitly specified in programs, access scopes will be lexically embedded with the most local definition applying when the same identifier appears at several levels. The language will use the above lexically embedded scope rules for declarations and other definitions of identifiers to make them easy to recognize and to avoid errors and ambiguities from multiple use of identifiers in a single scope.

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

A simple base alone is not sufficient for a common language. Even though in theory such a language provides the necessary power and the capability for specialization to particular applications, the users of the language cannot be expected to develop and support common libraries under individual projects There will be broad support for libraries common to users of well recognized application areas. Application libraries will be developed as early as possible.

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

The usefulness of a language derives primarily from the existence and accessibility of specialized application-oriented data and operations. Whether a library should contain source or object code is a question of implementation efficiency and should not be specified in the definition of the source language, but the source language description will always be available. It should be remembered, however, that interfaces cannot be validated at program assembly time without some equivalent of their source language interface specifications, that object modules are machine-dependent and, therefore, not portable, that source code is often more compact than object code, and that compilers for simple languages can sometimes

compile faster than a loader can load from relocatable object programs. Library routines written on other languages will not be prohibited provided the foreign routine has object code compatible with the calling mechanisms used in the Common HOL and providing sufficient header information (e.g., parameter types, form and number) is given with the routine in Common HOL form to permit the required compile time checks at the interface.

F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

Compools have proven very useful in organizing and controlling shared data structures and shared routines. A similar mechanism will be available to manage and control access to related library definitions.

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

The convenience, ease of use and savings in production and maintenance costs resulting from using high order languages come from being able to use specialized capabilities without building them from scratch. Thus, it is essential that high level capabilities be supplied with the language. The idea is not to provide all the many special cases in the language, but to provide a few general case . hich will cover the special cases.

There is currently little agreement on standard operating system, I/O, or file system interfaces. This does not preclude support of one or more forms for the near term. For the present the important thing is that one be chosen and made available as a standard supported library definition which the user can use with confidence.

G. CONTROL STRUCTURES
1. Kinds of Control Structures
2. The Go To
3. Conditional Control
4. Iterative Control
5. Routines
6. Parallel Processing
7. Exception Handling
8. Synchronization and Real Time

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

These mechanisms, hopefully, provide a spanning set of control structures. The most appropriate operations in several of these areas is an open question. For the present, the choice will be a spanning set of composable control primitives each of which is easily mapped onto object machines and which does not impose run time charges when it is not used. Whether parallel processing is real (i.e., by multiprocessing) or is synthesized on a single sequential processor, is determined by the object machine, but if programs are written as if there is true parallel processing (and no assumption about the relative speeds of the processors) then the same results will be obtained independent of the object environment.

It is desirable that the number of primitive control structures in the language be minimized, not by reducing the power of the language, but by selecting a small set of composable primitives which can be used to easily build other desired control mechanisms within programs. This means that the capabilities of control mechanisms must be separable so that the user need not pay either program clarity or implementation costs for undesired specialized capabilities. By these criteria, the Algol-60 "FOR" would be undesirable because it imposes the use of a loop control variable, requires that there be a single terminal condition and that the condition be tested before each iteration. Consequently, "FOR" cannot be composed to build other useful iterative control structures (e.g., FORTRAN "DO"). The ability to compose control structures does not imply an ability to define new control operations and such an ability to define new control operations, and such an ability is in conflict with the limited parameter passing mechanisms of C7.

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.

The "GO TO" is a machine level capability which is still needed to fill in any gaps which might remain in the choice of structured control primitives, to provide compatibility for translitterating programs written in older languages, and because of the wide familiarity of current practitioners with its use. The language should not, however, impose unnecessary costs for its presence. The "GO TO" will be limited to explicitly specified program labels at the same scope level. Neither should the language provide specialized facilities which encourage its use in dangerous and confusing ways. Switches, designational expressions, label variables, label parameters and numeric labels are not desired. Switches here refer to the unrestricted switches which are generalizations of the "GO TO" and not to case statements which are a general form for conditionals(see G3). This requirements should not be interpreted to conflict with the specialized form of control transfer provided by the exception handling control structure of G7.

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

The conditional control operations will be fully partitioned (e.g., an "ELSE" clause must follow each "IF THEN") so the choice is clear and explicit in each case. There will be some general form of conditional which allows an arbitrary computation to determine the selected situation (e.g., Zahn's device(Donald E. Knuth, "Structured Programming with go to Statements," ACM Computer Surveys, Vol. 6, No. 4, December 1974) provides a good solution to the general problem). Special mechanisms are also needed for the more common cases of the Boolean expression (e.g., "IF THEN ELSE") and for value or type discrimination (e.g., "CASE" on one of a set of values or subtype of a union).

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables t be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

In its most general form, a programmed loop is executed repetitively until some computed predicate becomes true. There may be more than one terminating predicate, and they might appear anywhere in the loop. Specialized control structures (e.g., "WHILE DO") have been used for the common situation in which the

termination conditions precedes each iteration. The most common case is termination after a fixed number of iterations and a specialized control structure should be provided for that purpose (e.g., FORTRAN "DO" or Algol-60 "FOR"). A problem which arises in many programming languages is that loop control variables are global to the iterative control and thus, will have a value after loop termination, but that value is usually an accident of the implementation. Specifying the meaning of control variables after loop termination in the language definition resolves the ambiguity but must be an arbitrary decision which will not aide program clarity or correctness, and may interfere with the generation of efficient object code. Loop control variables are by definition variables used to control the repetitive execution of a programmed loop and as such will be local to the loop body, but at loop termination it will be possible to pass their value (or any other computed value) out of the loop, conveniently and efficiently.

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

Recursion is desirable in many applications because it contributes directly to their elegance and clairty and simplifies proof procedures. Indirectly, it contributes to the reliability and maintainability of some programs. Recursion is required in order to avoid unnecessarily opaque, complex and confusing programs when operating on recursive data structures. Recursion has not been widely used in DoD software because many programming languages do not provide recursion, practitioners are not familiar with its use, and users fear that its run time costs are to high. Of these, only the run time costs would justify its exclusion from the language.

A major run time cost often attributed to recursion is the need for the presence of a set of "display" registers which are used to keep track of the addresses of the various levels of lexically imbedded environments and which must be managed and updated at run time. The display, however, is necessary only in programs in which routines access variables which are global to their own definition, but local to a more global recursive procedure. This possibility can easily be removed by prohibiting the definition of procedures within the body of a recursive procedure. The utility of such a combination of capabilities is very questionable, and this single restriction will eliminate all added execution costs for nonrecursive procedures in programs which contain recursive procedures.

As with any other facility of the language, routines should be implemented in the most efficient manner consistent with their use and the language should be designed so that efficient implementations are possible. In particular, the most possible regardless of whether the language or even the program contains recursive

procedures. When any routine makes a procedure call as its last operation before exit (and this is quite common for recursive routines) the implementation might use the same data area for both routines, and do a jump to the head of the called procedure thereby saving much of the overhead of a procedure call and eliminating a return. The choice between recursive and nonrecursive routines involves trade-offs. Recursive routines can aid program clarity when operating on recursive data, but can detract from clarity when operating on iterative data. They can increase execution time when procedure call overhead is greater than loop overhead and can decrease execution times when loop overhead is the more expensive. Finally, program storage for recursive routines is often only a small fraction of that for a corresponding iterative procedure, but the data storage requirements are often much larger because of the simultaneous presence of several activations of the same procedure.

G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

A parallel processing capability is essential in embedded computer applications. Programs must send data to, receive data from, and control many devices which are operating in parallel. Multiprogramming (a form of pseudo paralell processing) is necessary so that many programs within a system can meet their differing real time constraints. The parallel processing capability will minimally provide the ability to define and call parallel processing and the ability to gain exclusive use of system resources in the form of data structures, devices and pseudo devices. This latter ability satisfies one of the two needs for synchronization of parallel processes. The other is required in conjunction with real time constraints (see G8).

The parallel processing capability will be defined as true parallel (as opposed to coroutine) primitives, but with the understanding that in most implementations the object computer will have fewer processors (usually one) than the number of parallel paths specified in a program. Interleaved execution in the implementation may be required.

The parallel processing features of the language should be selected to eliminate any unnecessary overhead associated with their use. The costs of parallel processes are primarily in run time storage management. As with recursive routines most accessing and storage management problems can be eliminated by prohibiting complex interactions with other language facilities where the combination has little if any utility. In particular, it will not be possible to define a parallel routine within the

body of a recursive routine and it will not be possible to define any routine including parallel routines within the body of those parallel routines which can have multiple simultaneous activations. If the language permits several simultaneous activations of a given parallel process then it might require the user to give a upper bound on the number which can exist simultaneously. The latter requirement is reasonable for parallel processes because it is information known by the programmer and necessary to the maintainer, because parallel processes cannot normally be stacked, and because it is necessary for the compilation of efficient programs.

G7. The exception handing control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

It is essential in many aplications that there be no program halts beyond the user's control. The user must be able to specify the action to be taken on any exception situation which might occur within his program. The exception handling mechanism will be parameterized so data can be passed to the recovery point. Exception situations might include arithmetic overflow, exhaustion of available space, hardware errors, any user defined exceptions and any run time detected programming error.

The user will be able to write programs which can get out of an arbitrary nest of control and intercept it at any embedding level desired. The exception handling mechanism will permit the user to specify the action to be taken upon the occurrence of a designated exception within any given access scope of the program. The transfers of control will, at the users option, be either forward in the program (but never to a narrower scope of access or out of a procedure) or out of the current procedure through its dynamic (i.e., calling structure. The latter form requires an exception handling formal parameter class (see C7).

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

When parallel or pseudo parallel paths appear in a program it must be possible to specify their relative priorities and to synchronize their executions. Synchronization can be done either through exclusive access to data (see G6) or through delays terminated by designated situations occurring within the program.

These situations should include the elapse of program specified time intervals, occurrence of hardware interrupts and those designated in the program. There will be no implicit evaluation of program determined situations. Time delays will be program specifiable for both real and simulated times.

H. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics
2. No Syntax Extensions
3. Source Character Set
4. Identifiers and Literals
5. Lexical Units and Lines
6. Key Words
7. Comment Conventions
8. Unmatched Parentheses
9. Uniform Referent Notation
10. Consistency of Meaning

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

Clarity and readability of programs will be the primary criteria for selecting a syntax. Each of the above points can contribute to program clarity. The use of free format, mnemonic identifiers and conventional forms allows the programmer to use notations which have their familiar meanings to put down his ideas and intentions in the order and form that humans think about them, and to transfer skills he already has to the solution of the problem at hand. A simple uniform language reduces the number of cases which must be dealt with by anyone using the language. If programs are difficult for the translator to parse they will be difficult for people. Similar things should use the same notations with the special case processing reserved for the translator and object machine. The purpose of mnemonic identifiers and key words is to be informative and increase the distance between lexical units of programs. This does not prevent the use of short identifiers and short key words.

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences.

If the user can change the syntax of the language, then he can change the basic character and understanding of the language. The distinction between

semantic extensions and syntactic extensions is similar to that between being able to coin new words in English or being able to move to another natural language. Coining words requires learning those new meanings before they can be used, but at the same time increases the power the language for some application areas. Changing the grammar, (e.g., Franglais, the use of French grammar with interspersed English words) however, undermines the basic understanding of the language itself, changes the mode of expression, and removes the commonalities which obtain between various specializations of the language. Growth of a language through definition of new data and operations and the introduction of new words and symbols to identify them is desirable, but there should be no provision for changing the grammatical rules of the language. This requirement does not conflict with E4 and does not preclude associating new meanings with existing infix operators.

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

A common language should use notations and a character set convenient for communicating algorithms, programs, and programming techniques among its users. On the other hand, the language should not require special equipment (e.g., card readers and printers) for its use. The use of the 64 character ASCII subset will make the language compatible with the federal information processing standard 64 character set, FIPS-1, which has been adopted by the U.S.A. Standard code for Information Interchange (USASCII). The language definition will specify the translation from the publication language into the restricted character set.

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

Lexical units of the language should be defined in a simple uniform and easily understood manner. Some possible break characters are the space(W. Dijkstra, coding examples in Chapter I, "Notes in Structured Programming," in Structured Programming by O.-J. Dahl, E. W. Dijkstra and C.A.R. Moore, Academic Press, 1972. & Thomas A. Standish, "A Structured Program to Play Tic-Tac-Toe," notes for Information and Computer Science 3 course at Univ. of California-Irvine, October 1974) (i.e., any number of spaces or end-of-line), the underline and the tilde. The space cannot be used if identifiers and user defined infix operators are lexically indistinguishable, but in such a case the formal grammar for the language would be ambiguous (see H1). A literal break character contributes to the readability of

programs and makes the entry of long literals less error prone. With a space as a break character one can enter multipart (i.e., more than one lexical unit) identifiers such as "REAL TIME CLOCK" or long literals, such as, "3.14159 26535 89793." Use of a break can also be used to guarantee that missing quote brackets on character literals do not cause errors which propagate beyond the net end-of-line. the language should require separate quoting of each line of a long literal: "This is a long" "literal string".

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

Many elementary input errors arise at the end of lines. Programs are input on line oriented media but the concept of end-of-line is foreign to free format text. Most of the error prone aspects of end-of-line can be eliminated by not allowing lexical units to continue over lines. The sometimes undesirable effects of this restriction can be avoided by permitting identifiers and literals to be composed from more than one lexical unit (see H4) and by evaluating constant expressions at compile time (see C4).

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

By key words of the language are meant those symbols and strings which have special meaning in the syntax of programs. They introduce special syntactic forms such as are used for control structures and declarations or the are used as infix operators, or as some form of parenthesis. Key words will be reserved, that is unusable as identifiers, to avoid confusion and ambiguity. Key words will be few in number because each new key word introduces another case in the parsing rules and thereby adds to complexity in understanding the language, and because large numbers of key words inconvenience and complicate the programmer's task of choosing informative identifiers. Key words should be concise, but being information is more important than being short. A major exception is the key word introducing a comment; it is the comment and not its key word which should do the informing. Finally, there will be no place in a source language program in which a key word can be used in place of an identifier. That is, functional form operations and special data items built into the language or accessible as a standard extension will not be treated as key words but will be treated as any other identifier.

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

These are all obvious points which will encourage the use of comments in programs and avoid their error prone features in some existing languages. Comments anywhere reasonable in a program will not be taken to mean that they can appear internal to a lexical unit, such as, an identifier, key word, or between the opening and closing brackets of a character string. One comment convention which nearly meets these criteria is to have a special quote character which begins comments and with either the quote or an end-of-line ending each comment. This allows both embedded and line-oriented comments.

H8. The language will not permit unmatched parentheses of any kind.

Some programming languages permit closing parentheses to be omitted. If, for example, a program contained more "BEGINs" than "ENDs" the translator might insert enough "ENDs" at the end of the program to make up the difference. This makes programs easier to write because it sometimes saves writing several "ENDs" at the end of programs and because it eliminates all syntax errors for missing "ENDs." Failure to require proper parentheses matching makes it more difficult to write correct programs. Good programming practice requires that matching parentheses be included in programs whether or not they are required by the language. Unfortunately, if they are not required by the language then there can be no syntax check to discover where errors were made. The language will require full parentheses matching. This does not preclude syntactic features such as "case x of s1, s2...sn end case" in which "end" is paired with a key word other than "begin." Nor does it alone prohibit open forms such as "if-then-else-."

H9. There will be a uniform referent notation.

The distinction between function calls and data reference is one of representation, not of use. Thus, there will be no language imposed syntactic distinction between function calls and data selection. If, for example, a computed function is replaced by a lookup table there should be no need to change the calling program. This does not preclude the inclusion of more than one referent notation.

H10. No language defined symbols appearing in the same context will have essentially different meanings.

This contributes to the clarity and uniformity of programs, protects against psychological ambiguity and avoids some error prone features of extant languages. In particular, this would exclude the use of = to imply both assignment and equality, would exclude conventions implying that parenthesized parameters have special semantics (as with PL/1 subroutines), and would exclude the use of an assignment operator for other than assignment (e.g., left hand side function calls). It would not, however, require different operator symbols for integer, real or even matrix arithmetic since these are in fact special cases of the same abstract operations and would allow the use of generic functions applicable to several data types.

# I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic
2. Object Representation Specifications Optional
3. Compile Time Variables
4. Conditional Compilation
5. Simple Base Language
6. Translator Restrictions
7. Object Machine Restrictions

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

The only alternative is implementation dependent defaults with the translator determining the meaning of programs. What a program does, should be determinable from the program and the defining documentation for the programming language. This does not require that binding of all program properties be local to each use. Quite the contrary, it would, for example, allow automatic definition of assignment for all variables or global specification of precision. What it does require is that each decision be explicit in the language definition, global to some scope, or local to each use. Omission of any selection which affects the program logic will be treated as an error by the translator.

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

The language should be oriented to provide a high degree of management control and visibility to programs and toward self-documenting programs with the programmer required to make his decisions explicit. On the other hand, the programmer should not be forced to overspecify his programs and thereby cloud their logic, unnecessarily eliminate opportunities for optimization, and misrepresent arbitrary choices as essential to the program logic. Defaults will be allowed, in fact, encouraged in don't care situations. Such defaults will include data representations (see J4), open vs. closed subroutine calls (see J5), and reentrant vs. nonreentrant code generation.

I3. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

When a language has different host and object machines and when its compilers can produce code for several configurations of a given machine, the programmer should be able to specify the intended object machine configuration. The user should have control over the compile time variables used in his program. Typically they would be associated with the object computer model, the memory size, special hardware options, the operating system if present, peripheral equipment or other aspects of the object machine configuration. Compile time variables will be set outside the program, but available for interrogation within the program (see I4 and C4).

I4. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

An environmental inquiry capability permits the writing of common programs and procedures which are specialized at compile time by the translator as a function of the intended object machine configuration or of other compile time variables (see I3). This requirement is a special case of evaluation of constant expressions at compile time (see C4). It provides a general purpose capability for conditional compilation.

I5. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capablity not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

The capabilities available in any language can be partitioned into two groups, those which are definable within the base and those which provide an essential primitive capability of the language. The smaller and simpler the base the easier the language will be to learn and use. A clearly delineated base with features not in the base defined in terms of the base, will improve the ease and efficiency of learning, implementing and maintaining the language. Only the base need be implemented to make the full source language capability available.

Base features will provide relatively low level general purpose capabilities not yet specialized for particular applications. There will be no prohibition on a translator incorporating speciaized optimizations for particular extensions. Any extension provided by a translator will, however, be definable within the base language using the built-in definition facilities. Thus, programs using the extension will be translatable by any compiler for the language but not necessarily with the same object efficiency.

16. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

Limits on the number of array dimensions, the length of identifiers, the number of nested parentheses levels in expressions, or the number of identifiers in programs are determined by the translator and not by the object machine. Ideally, the limits should be set so high that no program (save the most abrasive) encounters the limits. In each case, however: (a) some limit must be set, (b) whatever the limit, it will impose on some and therefore must be known by the users of the translator, (c) letting each translator set its own limits means that programs will not be portable, (d) setting the limits very high requires that the translator be hosted only on large machines and (e) quite low limits do not impose significantly on either the power of the language or the readability of programs. Thus, the limits should be set as part of the language definition. They should be small enough that they do not dominate the compiler and large enough that they do not interfere with the usefulness of the language. If they were set at say the 99 percent level based on statistics from existing DoD computer programs the limits might be a few hundred for numbers of identifiers and less than ten in the other cases mentioned above.

17. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

Limits on the amount of run time storage, access to specialized peripheral equipments, use of special hardware capabilities and access to real time clocks are dependent on the object machine and configuration. The translator will report when a program exceeds the resources or capabilities of the intended object machine but will not build in arbitrary limits of its own.

## J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

1. Efficient Object Code
2. Optimizations Do Not Change Program Effect
3. Machine Language Insertions
4. Object Representation Specifications
5. Open and Closed Routine Calls

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

The base language and library definitions might contain features and capabilities which are not needed by everyone, or at least, not be everyone all the time. The language should not force programs to require greater generality than they need. When a program does not use a feature or capability it should pay no run time cost for the feature being in the language or library. When the full generality of a feature is not used, only the necessary (reduced) cost should be paid. Where possible, language features (such as, automatic and dynamic array allocation, process scheduling, file management and I/O buffering) which require run time support packages should be provided as standard library definitions and not as part of the base language . The user will not have to pay time and space for support packages he does not use. Neither will there be automatic movement of programs or data between main store and backing shore which is not under program control (unless the object machine has virtual memory with underlying management beyond the control of all its users). Language features will result in special efficient object codes when their full generality is not used. A large number of special cases should compile efficiently. For example, a program doing numerical calculations on unsubscripted real variables should produce code no worse than FORTRAN. Parameter passing for single argument routines might be implemented much less expensively than multiple argument routines.

One way to reduce costs for unneeded capabilities is to have a base language whose data structures and operations provide a single capability which is composable and has a straight-forward implementation in the object code of conventional architecture machines. If the base language components are easily composable they can be used to construct the specialized structures needed by specific applications, if they are simple and provide a single capability they will not force the use of unneeded capabilities in order to obtain needed capabilities, and if they are compatible with the features normally found in sequential uniprocessor digital computers with random access memory they will have near minimum or at least low cost implementation on many object machines.

J2. Any optimizations performed by the translator will not change the effect of the program.

More simply, the translator cannot give up program reliability and correctness, regardless of the excuse. Note that for most programming languages there are few known safe optimizations and many unsafe ones. The number of applicable safe optimizations can be increased by making more information available to the compiler and by choosing language constructs which allow safe optimizations. This requirement allows optimization by code motion providing that motion does not change the effect of the program.

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

It is difficult to be enthusiastic about machine language insertions. They defeat the purpose of machine independence constrain the implementation techniques complicate the diagnostics, impair the safety of type checking, and detract from the reliability, readability, and modifiability of programs. The use of machine language insertions is particularly dangerous in multiprogramming applications because they impair the ability to exclude, "a priori," a large class of time-dependent bugs. Rigid enforcement of scope rules by the compiler in real time applications is a powerful tool to ensure that one sequential process will not interfere with others in an uncontrolled fashion. Similarly, when several independent programs are executed in an interleaved fashion, the correct execution of each may depend on the others not having improperly used machine language insertions.

Unfortunately machine language insertions are necessary for interfacing special purpose devices, for accessing special purpose hardware capabilities, and for certain code optimizations on time critical paths. Here we have an example of Dijkstra's dilemma in which the mismatch between high level language programming and the underlying hardware is unacceptable and there is no feasible way to reject the hardware. The only remaining alternative is to "continue bit pushing in the old way, with all the known ill effects." Those ill effects can, however, be constrained to the smallest possible perimeter in practice if not in theory. The ability to enter machine language should not be used as an excuse to exclude otherwise needed facilities from the HOL; the abstract description of programs in the HOL should not require the use of machine language insertions. The semantics of machine language insertions will be determinable from the HOL definition and the object machine description alone and not dependent on the translator characteristics Machine language insertions will be encapsulated so they can be easily recognized and so that it is clear which variables and program identifiers are accessed within the insertion. The machine language insertions will be permitted only within the body of compile

time conditional statements (see I4) which depend on the object machine configuration (see I3). They will not be allowed interspersed with executable statements of the source language.

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

It is often necessary to give detailed specifications of the object data representations to obtain maximum density for large data files to meet format requirements imposed by the hardware of peripheral equipment, to allow special optimizations on time critical paths, or to ensure compatibility when transferring data between machines.

It will be possible to specify the order of the fields, the width of fields, the presence of don't care fields, and the position of word boundaries. It will be possible to associate source language identifiers (data or program) with special machine addresses. The use of machine dependent characteristics of the object representation will be restricted as with machine dependent code (see J3). When multiple fields per word are specified the compiler may have to generate some form of shift and mask operations for source program references and assignments to those variables (i.e., fields). As with machine-language insertions, object data specifications should be used sparingly and the language features for their use must be Spartan, nor grandiose.

If the object representation of a composite data object is not specified in the source program, there will be no specific default guaranteed by the translator. The translator might, for example, attempt to minimize access time and/or memory space in determining the object representation. It might, depending on the object machine characteristics, assign variables and fields of records to full words, but assign array elements to the smallest of bits, bytes, half words, words or exact multiple words permitted by the logical description.

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.

The use of inline open procedures can reduce the run time execution costs significantly in some cases. There are the obvious advantages in eliminating the parameter passing, in avoiding the saving of return marks, and in not having to pass arguments to and from the routine. A less obvious, but often more important advantage in saving run time costs is the ability to execute constant portions of routines at compile time and, thereby, eliminate time and space for those portions of the procedure body at run time. Open routine capability is especially important for machine language insertions.

The distinction between open and closed implementation of a routine is an efficiency consideration and should not affect the function of the routine. Thus, an open routine will differ from a syntax macro in that (a) its global environment is that of its definition and not that of its call and (b) multiple occurrences of a formal value (i.e., read only) parameter in the body have the same value. If a routine is not specified as either open or closed the choice will be optimal as determined by the translator.

K. PROGRAM ENVIRONMENT
1. Operating System Not Required
2. Program Assembly
3. Software Development Tools
4. Translator Options
5. Assertions and Other Optional Specifications

K1. The language will not require that the object machine have an operating system. When the object machine does have an operating system or executive program, the hardware/operating system combination will be interpreted as defining an abstract machine which acts as the object machine for the translator.

A language definition cannot dictate the architecture of existing object machines whether defined entirely in hardware or in a hardware/software combination. It can provide a source language representation of all the needed capabilities and attempt to choose these so they have an obvious and efficient translation in the object machines.

K2. The language will support the integration of separately written modules into an operational program.

Separately written modules in the form of routines and type definitions are necessary for the management of large software efforts and for effective use of libraries. The user will be able to cause anything in any accessible library to be inserted into his program. This is a requirement for separate definition but not necessarily for separate compilation. The decision as to whether separately defined program modules are to be maintained in source or object language form is a question of implementation efficiency, will be a local management option and will not be imposed by the language definition. The trade-offs involved are complicated by other requirements for type checking of parameters (see C6), for open subroutines (see J5), for efficient object representations (see J1), and for constant expression evaluation at compile time (see C4). In general, separate compilation increases the difficulty and expense of the interface validations needed for program safety and reliability and detracts from object program efficiency by removing many of the optimizations otherwise possible at the interfaces, but at the same time it reduces the cost and complexity of compiiation.

K3. A family of programming tools and aids in the form of support packages including linkers, loaders and debugging systems will be made available with the language and its translators. There will be a consistent easily used user interface for these tools.

The time has passed in which a programming language can be considered in isolation from its programming environment. The availability of programming tools which need not be developed and/or supported by individual projects is a major factor in the acceptability of a language. There is no need to restrict the kinds or form of support software available in the programming environment, and continued development of new tools should be encouraged and made available in a competitive market. It is, however, desirable that tools be developed in their own source language to simplify their portability and maintainability.

K4. A variety of useful options to aid generation, test, documentation and modification of programs will be provided as support software available with the language or as translator options. As a minimum these will include program editing, post- mortem analysis and diagnostics, program reformating for standard indentations, and cross-reference generation.

There will be special facilities to aid the generation, test, documentation and modification of programs. The "best" set of capabilities and their proper form is not currently known. Since nonstandard translator options and availability of nonstandard software tools and aids do not adversely affect software commonality, the language definition and standards will not dictate arbitrary choices. Instead, the development of language associated tools and aids will be encouraged within the constraint of implementing and supporting the source language as defined. Tools and debugging aids will be source language oriented.

Some of the translator options which have been suggested and may be useful include the following. Code might be compiled for assertions which would give run time warnings when the value of the assertion predicate is false. It might provide run time tracing of specified program variables. Dimensional analysis might be done on units of measure specifications. Special optimizations might be invoked. There might be capability for timing analysis and gathering run time statistics. There might be translator supplied feedback to provide management visibility regarding progress and conformity with local conventions. The user might be able to inhibit code generation. There might be facilities for compiling program patches and for controlling access to language features. The translator might provide a listing of the number of instructions generated against corresponding source inputs and/or an estimate of their execution times. It might provide a variety of listing options.

K5. The source language will permit inclusion of assertions, assumptions, axiomatic definitions of data types, debugging specifications;, and units of measures in programs. Because many assertional methods are not yet powerful enough for practical use, nor sufficiently well developed for standardization, they will have the status of comments.

There are many opinions on the desirability, usefulness, and proper form for each of these specifications. Better program documentation is needed and specifications of these kinds may help. Specifications also introduce the possibility of automated testing, run time verification of predicates, formal program proofs, and dimensional analysis. The language will not prohibit inclusion of these forms of specification if and when they become available for practical use in programs. Assertions, assumptions, axiomatic definitions and units of measure in source language programs should be enclosed in special brackets and should be treated as interpreted comments -- comments which are delimited by special comment brackets and which may be interpreted during translation or debugging to provide units analysis, verification of assertions and assumptions, etc.--but whose interpretation would be optional to translator implementations.

L. TRANSLATORS
1. No Superset Implementations
2. No Subset Implementations
3. Low-Cost Translation
4. Many Object Machines
5. Self-Hosting Not Required
6. Translator Checking Required
7. Diagnostic Messages
8. Translator Internal Structure
9. Self-Implementable Language

L1. No implementation of the language will contain source language features which are not defined in the language standard. Any interpretation of a language feature not explicitly permitted by the language definition will be forbidden.

This guarantees that use of programs and software subsystems will not be restricted to a particular site by virtue of using their unique version of the language. It also represents a commitment to freezing the source language, inhibiting innovations and growth in the form of the source language, and confining the base language to the current state of the art in return for stability, wider applicability of software tools, reusable software, greater software visibility, and increased payoff for tool building efforts. It does not, however, disallow library definition optimizations which are translator unique.

L2. Every translator for the language will implement the entire base language. There will be no subset implementations of the base language.

If individual compilers implement only a subset of the language, the there is no chance for software commonality. If a translator does not implement the entire language, it cannot give its users access to standard supported libraries or to application programs implemented on some other translator. Requiring that the full language be implemented will be expensive only if the base language is large, complex, and nonuniform. The intended source language product from this effort is a small simple uniform base language with the specialized features, support packages, and complex features relegated to library routines not requiring direct translator support. If simple low cost translators are not feasible for the selected language, then the language is too large and complex to be standardized and the goal of language commonality will not be achievable.

L3. The translator will minimize compile time costs. A goal of any translator for the language will be low cost translation.

Where practical and beneficial the user will have control over the level of optimization applied to his programs. The programmer will have control over the tradeoffs between compile time and run time costs. The desire for small efficient translators which can be hosted by machines with limited size and capability should influence the design of the base language against inclusion of unnecessary features and towards systematic treatment of features which are included. The goal will be effective use of the available machines both in object execution and translation and not maximal speed of translation.

Translation costs depend not only on the compiler but the language design. Both the translator and the language design will emphasize low cost translation, but in an environment of large and long-lived software products this will be secondary to requirements for reliability and maintainability. Language features will be chosen to ensure that they do not impose costs for unneeded generality and that needed capabilities can be translated into efficient object representations. This means that the inherent costs of specific language features is the context of the total language must be understood by the designers, implementers and users of the language. One consequence should be that trivial programs compile and run in trivial time. On the other hand, significant optimization is not expected from a minimal cost translator.

L4. Translators will be able to produce code for a variety of object machines. The machine independent parts of translators might be built independent of the code generators.

There is currently no common widely used computer in the DoD. There are at least 250 different models of commercial machines in use in DoD with many more specialized machines. A common language must be applicable to a wide variety of models and sizes of machines. Translators might be written so they can produce object code for several machines. This reduces the proliferation of translators and makes the full power of an existing translator available at the cost of producing an additional code generator.

L5. The translator need not be able to run on all the object machines. Self-hosting is not required, but is often desirable.

The DoD operational programming environment includes many small machines which are unable to support adequately the design, documentation, test, and

debugging aids necessary for the development of timely, reliable or efficient software. Large machine users should not be penalized for the restrictions of small machines when a common language is used. On the other hand, the size of machines which can host translators should be kept as small as possible by avoiding unnecessary generality in the language.

L6. The translator will do full syntax checking, will check all operations and parameters for type compatibility and will verify that all language imposed semantic restrictions on the source programs are met. It will not automatically correct errors detected at compile time.

The purpose of source language redundancy and avoidance of error prone language features is reliability. The price is paid in programmer inconvenience in having to specify his intent in greater detail. The payoff comes when the translator checks that the source program is internally consistent and adheres to its authors' stated intentions. There is a clear trade-off between error avoidance and programming ease; surveys conducted in the Services show that the programmers as well as managers will opt for error avoidance over ease when given the choice. The same choice is dictated by the need for well documented maintainable software.

L7. The translator will produce compile time explanatory diagnostic error and warning messages. A suggested set of error and warning situations will be provided as part of the language definition.

The translator will attempt to provide the maximal useful feedback to its user. Diagnostic messages will not be coded but will be explanatory and in source language terms. Translators will continue processing and checking after errors have been found but should be careful not to generate erroneous messages because of translator confusion. The translator will always produce correct code; when source programs errors are encountered by the translator or referenced program structures omitted, the compiler will produce code to cause a run time exception condition upon any attempt to execute those parts of the program. Warnings will be generated when a source language construct is exceptionally expensive to implement on the specified object machine. A suggested set of diagnostic messages provided as part of the language definition contributes to commonality in the implementation and use of the language. The discipline of designing diagnostic messages keyed to the design may also uncover pitfalls in the language design and thereby contribute to a more precise and better understood language description.

L8. The characteristics of translator implementations will not be dictated by the language definition or standards.

The adoption of a common language is a commitment to the current state of the art for programming language design for some duration. It does not, however, prevent access to new software and hardware technology, new techniques and new management strategies which do not impact the source language definition. In particular, innovation should be encouraged in the development of translators for a common language providing they implement exactly the source language as defined. Translators like all computer programs should be written in expectation of change.

L9. Translators for the language will be written in their own source language.

There will be at least one implementation of the translator in its own language which does all parsing and compile-time checking and produces an output suitable for easy translation to specific object machines. If the language is well-defined and uniform in structure, a self-description will contribute to understanding of the language. The availability of the machine independent portion of a translator will make the full power of the language available to any object machine at the cost of producing an additional code generator (whose cost may be high) and it reduces the likelihood of incompatible implementations. Translators written in their own source language are automatically available on any of their object machines providing the object machine has sufficient resources to support a compiler.

M. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only
2. Unambiguous Definition
3. Language Documentation Required
4. Control Agent Required
5. Support Agent Required
6. Library Standards and Support Required

M1. The language will be composed from features which are within the state of the art and any design or redesign which is necessary to achieve the needed characteristics will be conducted as an engineering design effort and not as a research project.

The adoption of a common language can be successful only if it makes available a modern programming language compatible with the latest software technology and is compatible with "best" current programming practice but the design and implementation of the language should not require additional research or require use of untried ideas. State-of-the-art cannot, however, be taken to mean that a feature has been incorporated in an operational DoD language and used for an extended period, or DoD will be forever tied to the technology of FORTRAN-like languages; but there must be some assurances through analysis and use that its benefits and deficiencies are known. The larger and more complex the structure, the more analysis and use that should be required. Language design should parallel other engineering design efforts in that it is a task of consolidation and not innovation. The language designer should be familiar with the many choices in semantic and syntactic features of language and should strive to compose the best of these into a consistent structure congruous with the needed characteristics. The language should be composed from known semantic features and familiar notations, but the use of proven feature should not necessarily impose that notation. The language must not just be a combination of existing features which satisfy the individual requirements but must be held together by a consistent and uniform structure which acts to minimize the number of concepts, consolidates divergent features and simplifies the whole.

M2. The semantics of the language will be defined unambiguously and clearly. To the extent a formal definition assists in attaining these objectives, the language's semantics will be specified formally.

A complete and unambiguous definition of a common language is essential. Otherwise each translator will resolve the ambiguities and fill in the gaps in its own

unique way. There are currently a variety of methods for formal specification of programming language semantics but it remains a major effort to produce a rigorous formal description and the resulting products are of questionable practical value. The real value in attempting a formal definition is that it reveals incomplete and ambiguous specifications. An attempt will be made to provide a formal definition of any language selected but success in that effort should not be requisite to its selection. Formal specification of the language might take the form of an axiomatic definition, use of the Vienna Definition Language, or use of some other formal semantic system.

M3. The user documentation of the language will be complete and will include both a tutorial introductory description and a formal in-depth description. The language will be defined as if it were the machine level language of an abstract digital computer.

The language should be intuitively correct and easily learned and understood by its potential users. The language definition might include an Algol-60 like description(P. Naur (Ed.), "Revised Report on the Algorithmic Language Algol-60," Communication of the A.C.M. Vol.6, No. 1, January 1963, p. 1-17.) with the source language syntax given in BNF or some other easily understood metalanguage and the corresponding semantics given in English. As with the descriptions of digital computer hardware, the semantics and syntax of each feature must be defined precisely and unambiguously. The action of any legal program will be determinable from the program and the language description alone. Any computation which can be described in the language will ultimately draw only on capabilities which are built into the language. No characteristics of the source language will be dependent on the idiosyncrasies of its translators.

The language documentation will include syntax, semantics and examples of each language construct, listings of all key words and language defined defaults. Examples shall be included to show the intended use of language features and to illustrate proper use of the language. Particularly expensive and inexpensive constructs will be pointed out. Each document will identify its purpose and prerequisites for its use.

M4. The language will be configuration managed throughout its total life cycle and will be controlled at the DoD level to ensure that there is only one version of the source language and that all translators conform to that standard.

Without controls a hopefully common language may become another umbrella under which new languages proliferate while retaining the same name. All compilers will be tested and certified for conformity to the standard specification and freedom from known errors prior to their release for use in production projects. The language manager will be on the OSD staff, but a group within the Military Departments or Agencies might act as the executive agent. A configuration control board will be instituted with user representation and chaired by a member of the OSD staff.

M5. There will be identified support agent(s) responsible for maintaining the translators and for associated design, development, debugging and maintenance aids.

Language commonality is an essential step in achieving software commonality, but the real benefits accrue when projects and contractors can draw on existing software with assurance that it will be supported, when systems can build from off the shelf components or at least with common tools, and when efforts can be spent to expand existing capabilities rather than building from scratch. Support of common widely used tools and aids should be provided independent of projects if common software is to be widely used. Support should be on a DoD-wide basis with final responsibility resting with a stable group or groups of qualified in-house personnel.

M6. There will be standards and support agents for common libraries including application-oriented libraries.

In a given application of a programming language three levels of the system must be learned and used: the base language, the standard library definitions used in that application area, and the local application programs. Users are responsible for the local application programs and local definitions but not for the language and its libraries which are used by many projects and sites. A principal user might act as agent for an entire application area.

# Appendix III

# REVIEW OF "TINMAN"

## Section I. INTRODUCTION

### 1. Overall Reaction.

At the outset, it must be emphasized that the requirements described by the "Tinman" are believed to be an excellent overall measure by which a candidate Common DoD Language can be assessed. Should the DoD obtain a language which fits these requirements closely, the quality of DoD software can be expected to increase significantly per unit cost. The comments given in this chapter are made with the objective of allowing the requirements of the Tinman to be further enhanced. While, of necessity, these comments suggest changes, they should be interpreted in the positive context stated above.

### 2. Notational Conventions Used in This Chapter.

The comments on the Tinman characteristics are classified as follows:

a. Ambiguity. The requirements are not clear in these areas. Several interpretations are possible.

b. Inconsistency. Two or more requirements are in conflict.

c. Overspecification. Although the requirements specify a reasonable feature, there are other features which seem to satisfy the intent of the requirement yet are ruled out by the specific details of the requirement.

d. Disagreement. These are requirements which we feel are not appropriate.

e. Comment. Additional comments which do not fit any of the above are included here.

## Section II. DATA AND TYPES

1. Typed Language (A1).

   a. Overspecification, Ambiguity.

      (1) The Tinman states that "the language will be typed" and that all types will be "determinable at compile time and unalterable at run-time". Types are considered properties of variables, expressions, etc. However, according to the Tinman, not all properties are included in the type. For example, according to A3, precision of reals does not distinguish type. According to A6, the upper bound for array subscripts can be determinable at run-time and is therefore not a property of an array type. It may be noted that two arrays which differ only in their upper bound will have the same type, yet one cannot be assigned to the other.

      (2) The ambiguity of these requirements is due to not carefully classifying those properties which distinguish types, and not specifying the effect of those properties which do not distinguish types.

      (3) The overspecification of requirements here is due to the partition of those properties which distinguish type from those that don't. This separation is somewhat arbitrary and a different partition does not necessarily result in languages with any differences in their behavior. The difference is strictly descriptive. A language that chooses a different partition than that reflected by the Tinman requirements will be in conflict with these requirements even when it completely satisfies the intent of the requirements.

      (4) We recommend that the requirements be changed as follows. First, rather than describe the requirements in terms of types, describe them in terms of properties of variables, expressions, etc. Second, the requirement should classify the properties by whether or not they must be known at

compile time and by what effect they have on behavior. Some properties (e.g., packing of representation) don't affect the value set at all. Other properties (e.g., precision, range, etc.) affect value sets but not the set of applicable operations. Still other properties (e.g., upper bound of arrays) do affect the set of valid operations. Finally, the requirement should be changed so that it discusses which classes of properties are permitted to be different in various operations (e.g., assignment, parameter passing, etc.).

b. Ambiguity, Inconsistency. All types "will be determinable at compile time". This requirement might be considered to be in conflict with C8, "specification of type will be optional in procedure declarations".

2. Data Types (A2).

Disagreement. The Tinman alleges that the listed types and type generators are sufficient "to efficiently mechanize any other desired type". It seems likely that realizations of pointers and "monitored variables", for example, would probably be quite difficult to achieve by these means. Consequently, we would include a typed-pointer type in the given list, and perhaps soften the stated claim a bit.

3. Precision (A3).

Disagreement. "Precision specification will apply to arithmetic operations." Although this may be acceptable for local variables with the same default precision as the operations, operations whose operands are non-local variables with a different default precision will use the wrong precision. It can be argued that most arithmetic with reals is likely to be performed with "single precision" values, and that the FORTRAN style of specifying only exceptions to this rule would suffice. Yet, the question must be raised as to whether the general trend towards explicit expression of intent ought not be adhered to in this context: namely, that each variable declaration should include specifications of its range and precision of values. For efficiency reasons, user-stated requirements of

precision will be adjusted upwards to correspond to hardware-provided single and double precision forms. Evaluation of operations, as in virtually all languages which recognize multiple precisions, will be conducted at the largest of the precisions of the operands. Requiring "a single specification of the precision" will cause programmers to specify the largest precision needed for any object in the scope. It is not clear that this contributes to readability, implementability, or quality of the software.

4. Fixed Point Numbers (A4).

a. Disagreement. "Fixed point numbers will be treated as exact quantities." For a given step size, not all operations will in general yield results with the same step size. For example, x := x/2 will be illegal. We recommend that fixed point numbers, like floating-point numbers, be treated as approximate quantities.

b. Disagreement. Two prominent reasons inflate the cost of software using fixed-point rather than floating-point arithmetic. First, the scaling rules, when automatically provided, are intricate, and difficult to use without error. Second, and more important, programmers must dedicate their attention to a consideration which is absent from floating-point calculations: the precision and scaling of intermediate values formed during the evaluation of expressions. Surprising losses of important bits of value can occur in these cases, which can be very difficult to locate; reliability thus drops, and debugging costs soar. We believe that automatic scaling of fixed-point values both encourages the use of fixed-point (which should be discouraged), and causes the introduction of subtle bugs through implicit actions invisible to the reader of the program. Consequently, automatic scaling should be minimized.

5. Character Data (A5).

Ambiguity. This requirement might be interpreted to mean that the lexical analyzer of a compiler must be extensible. This would introduce undesired complexity. This requirement should be clarified.

6. Arrays (A6).

a. Overspecification, Disagreement. "The number of dimensions, the [element] type, and the lower subscript bound [for the array] should be determinable at compile time." The specified distinction between upper and lower subscript bounds appears to offer idiosyncrasy without compensating advantage. We suggest that it would be better either 1) to require all integer array subscripts to begin at the same value (e.g., 1), or 2) to make the rules for determination of upper and lower bounds be identical, or 3) to write requirements so either of the solutions 1 or 2 is permitted.

b. Overspecification. This requirement implies that multidimensional arrays will be required. A similar effect can be achieved by composing multiple single dimensional arrays. We recommend that the requirements be changed to permit this alternate approach.

7. Records (A7).

a. Overspecification. This requirement specifies that overlaid data will be achieved via the record data structure. Some languages (e.g., CS-4) satisfy the intent but not the details of this requirement by having a separate discriminated union data structure.

b. Overspecification. The requirement states that the discrimination condition can be "any expression". We believe that restricted discrimination conditions (e.g., discrimination on the name of each overlay) do not significantly limit the power of the language. They may even result in more readable programs since the selected overlay is more apparent.

c. Comment. It should be noted here that Pascal provides such a discriminated union capability but with a mechanism that allows users to defeat type-rule enforcement as thoroughly as Fortran EQUIVALENCE does. To ensure the desired degree of safety, the desired degree of safety should be made part of the requirement.

## Section III. OPERATORS

1. Assignment and Reference (B1).

Disagreement. "Assignment and reference operations should be automatically defined for all data types." Assignment and reference, like other operations on data objects, should be subject to access regulation. The implementor of a "semaphore" data type, for example, will not wish any assignments to occur other than by means of "primitive" operations he makes available to semaphore users.

2. Equivalence (B2).

a. Ambiguity. Comparison of any two data objects. Four different terms or phrases are used in this section to describe the same comparison, resulting in serious ambiguity: "identity", "equivalence", "equality", and "multiple references to the same abstract data object".

b. Ambiguity. The text leaves a question as to whether it is intended that comparisons should be valid between objects of different types; the strong implication is that such comparisons would be permitted. We believe that a compile-time error message should result rather than a FALSE value, since the expression is likely to be a mistake.

c. Disagreement. "For floating-point numbers, identity will be defined as the same within the specified minimum precision." It is important to distinguish between the declared precision of a real variable and its effective precision as the result of a computation. In general the effective precision will be less than the declared precision. It would be beneficial for the requirements to specify a need for comparison operations which accept an additional argument specifying the precision to which the comparison is to be performed.

3. Relationals (B3).

a. Ambiguity. "Numbers and types defined by enumeration have an obvious ordering..." In an expression-oriented language, which is not precluded by the Tinman, there are contexts in which the enumeration order of a type is not obvious. If there is a canonical ordering for enumerated

types, this problem is resolved, but at the expense of eliminating order determined by the sequence in which values are written by the user in type definitions.

b. Overspecification. For user-defined types, the ordering of the resulting abstract type is not always the same as that of the representational type. We believe that the requirement should be that it is possible to define either ordered or unordered types. The inclusion of an automatic ordering together with an override constitutes an overspecification.

4. Arithmetic Operations (B4).

Overspecification. The requirements state that division will have a real result. This requirement is due to the fact that the integers are not closed under division. The Tinman-proposed answer for this problem is not the only reasonable solution, however. Equally reasonable alternatives are (a) to define integer division to produce an integer result by truncation, (b) to define integer division to be legal only for those operand values which produce an integer result (i.e., with a zero remainder), and (c) to disallow integer division entirely.

5. Boolean Operations (B6).

Inconsistency. "The operations and and or on scalars should be evaluated in short circuit mode." We agree with the requirement, as far as it goes. However, in accordance with E2, the capability for defining such operations should be made available to users. If and and or are built in, as implied, and if formal parameter binding classes are as restricted as in C7, the user does not have this capability.

6. Scalar Operations (B7).

a. Ambiguity. "The source language should permit scalar operations and assignment on conformable arrays and should permit data transfers between records or arrays of identical structure." There is a terminology change here whose intention is unclear: "data transfers" vs. "assignment and reference operations" (B1), and "structure" vs. "type" (also B1).

b. Ambiguity, Disagreement. The wording of the section clearly implies that it should be possible to add a (3 x 4) array of integers to a (2 x 3 x 2) array of integers. This requirement should stipulate instead that the arrays be of the same rank and that corresponding dimension sizes be equal.

7. Type Conversion (B8).

a. Ambiguity. It is not clear from this requirement whether mixed type operations (e.g., adding an integer to a real) are to be permitted. Mixed type operations can be considered to be either (a) operations which involve implicit conversions (prohibited by B8), or (b) different instances of a generic operation (permitted by C8).

b. Disagreement. "No conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter." There seems to be no strong reason for allowing this exception to the general rule against implicit type conversions. This exception can be a problem for parameters which are bound to variables (see C7). In this case the value of the formal parameter "constituent" can be changed, but not the particular constituent. This results in complex rules for assignment to the formal.

8. Changes in Numeric Representation (B9).

Inconsistency, Disagreement. Range checking is "optional for hardware installations which do not have overflow detection". This is in conflict with requirements B1 and B2 of Section V, which require that all implementations be the same. Even when hardware checking is not possible, software checks can always be performed. Since these software checks may have a severe impact on efficiency, their presence should be under the user's control (rather than under the implementer's control).

9. I/O Operations (B10).

Comment. We agree with the statements in B10, yet feel obliged to point out the troublesome lifetime issues raised by the need for these capabilities. Because the lifetime of data on files can exceed the lifetime of the processes or even programs in which the relevant type definitions occur,

and because files may be transferred to different installations (where different type definitions are in effect), there is a significant difficulty in performing rigorous type checking on filed data. Relaxation or elimination of type checking on files can lead to undetectable errors and to use of files as a means to defeat type checking.

10. Power Set Operations (B11).

Overspecification. The effect of a power set type can be achieved by permitting array indices to be enumeration types. An array of Booleans indexed by an enumeration type is quite similar to a power set. We recommend that this requirement be changed to include this alternate approach.

## Section IV. EXPRESSIONS AND PARAMETERS

1. Side Effects (C1).

Ambiguity, Comment. "Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left to right." The ambiguity here is that it is not stated what kinds of things are considered to be side effects. There are several possibilities, including (a) changes to variables that are (or could be) referenced in other arguments, (b) performing I/O, and (c) signalling exception conditions. If all of these (particularly c) are considered side effects, then the freedom of an optimizer to reorder argument evaluation is severely restricted. Possibly exception conditions should not be so severely restricted with respect to evaluation order.

2. Operand Structure (C2).

a. Comment. "Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader." This probably is possible only if explicit parentheses are required everywhere. Even the APL approach, with no operator hierarchy, fails to satisfy the "obvious"-requirement with every reader.

b. Ambiguity, Disagreement. "The user should not be able to define new operator precedence rules nor change the precedence of existing operators." We agree with this statement as worded; however, its intention is apparently to prohibit the definition of new operator precedence levels (not rules). We take strong exception to this intention. If operator hierarchy is supported at all (* being stronger than +, for example), it seems simply a caprice to prohibit all future designers of application packages from implementing operator sub-hierarchies meaningful to both their application area and the existing hierarchy for integers, Booleans, etc. If any restrictions of Tinman type are to be imposed, let them be uniformly applied to the "built-in" types as well, not just at the boundary between what we now know well (arithmetic and logical operations on built-in types) and what we have yet to exploit (user-defined types and operations). This boundary will change with experience: the language rules should not obstruct this experience.

3. Constant Expressions (C4).

Comment. "Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time." For the term "constant expression" to be meaningful, the language must clearly specify the rules for determining when an expression is "compile-time (or load-time) evaluable". The host/target machine differences here can be quite troublesome, requiring, in the limit, a target-machine simulator running on the host machine as part of the compiler. We recommend a cautious approach in this area so as not to burden "small" implementations with the necessity for elaborate pre-run-time evaluations of program segments.

4. Consistent Parameter Rules (C5).

a. Ambiguity. This requirement implies the existence of parameters to declarations. The meaning of this concept is obscure.

b. Inconsistency. "Uniformity and consistency contributes to ease of learning, implementing and using a language." This requirement together with requirement B6 which requires short circuit evaluation of certain Boolean operations, is in conflict with C7 which provides no parameter mechanism for user-defined short-circuited behavior.

6. Type Agreement in Parameters (C6).

Disagreement. "Formal parameters of a union type should be considered conformable to actual parameters of any of the component types." See the comment given for B8, above.

7. Formal Parameter Kinds (C7).

a. Disagreement. We strongly disagree with the oversimplified view of the formal parameter mechanism, one of the most key issues in modern programming language design for reliable software. We recommend that three classes of binding be provided for data: COPY, REF, and NAME. COPY and NAME correspond to classes supported in Algol 60, while REF is essentially from Fortran. Each should be viewed in a more general way than in their original environment: the intention of the procedure should be further detailed for

program readability and efficiency reasons. By specifying whether the procedure wishes to obtain the value of the actual parameter, modify the actual parameter, or both, the user enables the compiler to perform exactly appropriate validity checking on the correspondence between actual and formal parameters, and to avoid generation of code for transmission of data to or from the procedure when such transmission is not to be performed. While a skeptic can claim that this produces nine binding classes, such a claim obfuscates the fact that two separate properties are each being specified as one of three possibilities. This seems far superior to requiring that a large array be vulnerable to accidental alteration because it was received as a "read-write" parameter to avoid the space and time costs of making a local copy; or, to insisting that an integer be subjected to indirect addressing for many, many accesses by a procedure because its final value is to be returned as an output. NAME binding is exceptionally efficient when used with open expansion of procedure bodies, especially the type of open expansions which are necessary in the implementation of data abstractions. The reconsideration of the Tinman views in this area is urgently recommended.

b. Overspecification. "In addition there will be a formal parameter class for specifying the control actions when exception conditions occur...." This is only one of several possible ways of handling exception conditions. The requirements should be loosened to permit other equally attractive exception-handling mechanisms.

c. Disagreement, Overspecification. "In addition there will be a formal parameter class ... for procedure parameters." There is no strong reason why procedural parameters cannot be passed using one of the standard binding classes. We also disagree with this requirement. There seems to be no strong need for this facility which introduces considerable complexity to both the language and its implementation.

8. Formal Parameter Specifications (C8).

a. Ambiguity, Disagreement. "Specification of the type, range, precision, scale, and format of parameters will be optional." "...permits the writing of generic procedures...." "...eliminates the need for compile-time type parameters." The described capability seems to be that

of a substitution text-macro, which can do what text macros can do and no more. This falls far short of what is needed to properly implement generic procedures. Compile-time parameters and enquiries are essential.

b. Inconsistency. This requirement is inconsistent with A1, which requires that all types be known at compile time. See our comments for A1, above.

c. Ambiguity. The meaning of "generic" procedures is quite vague. There are at least two possible interpretations. First, the requirement might refer to a true "generic" procedure--a procedure that has several bodies, one of which is selected for each call based upon the types of the actual parameters. Another interpretation is that the requirement refers to a "polymorphic" procedure--a procedure that has a single body that uses generic operations. The meaning of a specific call is given by selecting the appropriate body for each of the referenced generic operations. (For example, a not-equal polymorphic operation can be implemented in terms of a Boolean not operation and a generic equal operation.) Both polymorphic and generic operations seem to be desirable features.

9. Variable Number of Parameters (C9).

a. Comment. "There should be provision for variable numbers of arguments, but in such cases all but a constant number of them should be of the same type." "There are many useful purposes for procedures with variable numbers of arguments. These include intrinsic functions such as print, ...." The described capability will be useful in implementing a function such as print only if the language restricts the members of its argument list to all being of the same type. Not even Fortran imposes such a confining restriction.

b. Disagreement. The advantages offered by permitting a variable number of arguments is more than offset by the added complexity introduced into the language. We recommend that this requirement be dropped.

## Section V. VARIABLES, LITERALS, AND CONSTANTS

1. Numeric Literals (D2).

Comment. "Literals are needed for all atomic data types...." Literals are also desirable for character strings. In fact, to complete the data type definition facilities, it would be meaningful to provide a method for prescribing literals of defined types. However, no known language has this ability, and we have no specific recommendation in this area at the moment. The "constructor" function notion is adequate in the meantime.

2. Numeric Range and Step Size (D4).

a. Ambiguity. "Range and step size specifications will not be interpreted as defining new types." See the comments under A1.

b. Inconsistency. "...Range specifications can not in general be evaluated at compile time." This requirement together with variable parameters (reference binding) (see C7) can have a serious impact on the efficiency requirements of J1.

3. Variable Types (D5).

a. Ambiguity, Disagreement. "The range of values... will be any built-in type." This implies that types are sets of values. We strongly disagree. See our comments under A1.

b. Inconsistency. "The range of values will be... a contiguous subsequence of any enumeration type." This is in conflict with E7 which states "Type definition by... subsetting [is] not desired".

4. Pointer Variables (D6).

a. Overspecification, Disagreement. "Consequently pointer/nonpointer will be a property only of variables for composite types and of composite array and record components." While the Tinman's desire for safety and control is correctly motivated, a more uniform approach, having the desired safety, is also possible. Pointer-type definitions can be required to designate the one type of

object to which they may point; in this way, no type defeating is possible, as it is with "absolute address" types. A heap storage class should also be supplied, allowing explicit allocation and de-allocation of objects to which pointers may point. Discriminated unions allow orderly realization of pointers which can point to objects of more than one type. No computational operations on pointers need be provided, ensuring that incorrect values are never generated.

b. Disagreement. "The use of pointers will be kept safe by prohibiting pointers to data structures whose allocation scope is narrower than that of the pointer variable." In dynamically modified list structures the lifetimes of individual "nodes" are such that this requirement cannot be met. Possible solutions include a heap garbage collector or a system of run-time checks. The garbage collector costs in time and space are frequently quite undesirable or intolerable in real-time applications. Run-time checks can be thoroughly effective, but there appear to be significant problems in combining programs which include such run-time checks with program segments (such as libraries) when the latter have been thoroughly debugged and the run-time checks removed. These problems arise only when checked and unchecked segments communicate via pointers, which is a relatively infrequent case.

## Section VI. DEFINITION FACILITIES

1. User Definitions Possible (E1).

Ambiguity. "The user of a language will be able to define new data types...." This requirement is very vague. What is a data type? (See our comments under A1.) Are new data types restricted to simply compositions of existing types and type generators, or is entirely new behavior possible? Do user-defined types have the same "power" as built-in types? Are representations to be hidden?

2. Consistent Use of Types (E2).

Ambiguity, Disagreement. "The 'use' of defined types will be indistinguishable from [that of] built-in types." The intent of this requirement is not clear. If it implies that a user-defined type can "simulate" the behavior of any built-in type, then realizing this requirement will be extremely difficult if not impossible with current technology.

3. Data Defining Mechanisms (E6).

Ambiguity, Disagreement. This seems to be a restriction on the types on which a newly-defined type can be based. It is overly restrictive. For example, it seems reasonable to permit a new user-defined type to depend upon a previously user-defined type. We recommend that this requirement be changed to permit any type to serve as the basis of a new type.

4. No Free Union of Subset Types (E7).

a. Ambiguity, Inconsistency. There is a conflict with D5 (see comments there).

b. Ambiguity. The meaning of free union is ambiguous. It could be either a variation on discriminated union or another name for equivalence.

## Section VII. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed (F1).

Ambiguity. It is not clear to what degree this requirement must be supported. Several possibilities exist. One possibility is to provide block structure with automatic variables. In languages having this feature, allocated variables may not be accessible in inner scopes if the same name is used for a local variable in that inner scope. Another possibility is to provide own (static) variables. Allocations of these variables persist longer than their scopes of access. A third possibility is to provide full separation. In this case, the scope of allocation is specified completely independent of the scope of access. The first case is supported by virtually all languages which are derivitives of ALGOL 60. The second case introduces a facility which is known to be error-prone and which is not needed in a language with user-defined types. The third case is a facility which is useful, but no known languages support such a facility without introducing considerable complexity.

2. Limiting Access Scope (F2).

Ambiguity, Overspecification, Comment. "Limited access specified in a type definition is necessary to guarantee that changes to data representations ... which do not affect calling programs are in fact safe." This intention seems to be that representations can be hidden in user-defined types. Limiting namescopes is not the only possible way of satisfying this intention. Also note that this condition is not sufficient to guarantee the ability to arbitrarily change a representation.

## Section VIII. CONTROL STRUCTURES

1. Kinds of Control Structures (G1).

a. Disagreement. "...selecting a small set of composable primitives which can be used to easily build other desired control mechanisms within programs." We question the advisability of this approach on four grounds:

(1) Without syntax extension (precluded by H2), a user is constrained by the existing notations (e.g., procedure invocation form) to contrive control structures of a funny-looking kind.

(2) It is unclear whether a user-defined control structure offers enough advantage to outweigh the attendant loss of program readability.

(3) Whether a comprehensive range of control structures can be achieved "by extension" at a tolerable level of inefficiency is probably still a research topic.

(4) The extension machinery itself will add, perhaps sizably, to the cost of compilers.

It seems preferable, at this time, to specify a reasonable set of control structures as primitives, and make no attempts to go further until a clear need can be demonstrated.

b. Ambiguity. Does the requirement for pseudo-parallel processing imply a coroutine-like mechanism?

2. The Go To (G2).

Ambiguity. "The 'GO TO' will be limited to... labels at the same scope level." Exactly what constitutes a scope level is subject to various interpretations: (a) a begin block (definitely!), (b) an alternative of an if statement (absolutely no), (c) a case statement (maybe), and (d) a repeat statement (maybe).

3. Conditional Control (G3).

Disagreement. The requirement that each IF statement require both a THEN and an ELSE clause is not justified. For readability, it is frequently useful to omit the ELSE clause and have control transfer to the next statement.

4. Iterative Control (G4).

a. Ambiguity, Overspecification. "The iterative control structure should permit the termination condition to appear anywhere in the loop...." Although this generalization of the WHILE and UNTIL statements appears useful, it must nevertheless be recognized that it can work in the context of nested loops only when the test for termination specifies explicitly the containing loop structure to which it belongs. It is unclear how the IF-statement with EXIT fails to give adequate capability in this respect.

b. Inconsistency. "...a specialized control structure should be provided for that purpose (e.g., FORTRAN DO or Algol-60 for)." This statement is in conflict with the sentence in G1 which states: "By these criteria, the Algol-60 for would be undesirable...".

c. Inconsistency, Overspecification. Whether or not a loop control variable should be local to the loop body, and whether its value should be specified and available following loop termination seem largely matters of taste and efficiency. If, as the Tinman states, "Specifying the meaning of control variables after loop termination in the language definition resolves the ambiguity but must be an arbitrary decision which will not aid program clarity or correctness...", then it is puzzling why in the following sentence it is asserted that "at loop termination it should be possible to pass their value...out of the loop".

d. Comment. It seems desirable to make an additional requirement that the value of a loop control variable may not be modified by any statements in the body of the loop.

5. Routines (G5).

Disagreement. "It will not be possible to define procedures written within the body of a recursive procedure." "A major...cost...is the need for...'display'

registers...." This restriction seems unnecessarily strong. Displays are not the only way of removing this restriction. Other techniques which limit run-time overhead to exactly those cases where this restriction is violated are possible. We recommend that this restriction be dropped.

6. Parallel Processing (G6).

Comment. The terminology used in this section does not agree with standard practice. For example, "simultaneous activations of a given parallel process".

7. Exception Handling (G7).

a. Comment. The motivation for the restriction of transfers of control "forward in the program" was not clear.

b. Overspecification. There are many reasonable alternative exception-handling mechanisms. The requirements here should be less specific.

8. Synchronization and Real Time (G8).

Overspecification. Asynchronous hardware interrupts are different from synchronous exception conditions. They need not necessarily be handled by the same mechanism.

## Section IX. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics (H1).

Inconsistency. "The source language... will be based on conventional forms." This may be in conflict with H10: "No language defined symbols... will have essentially different meanings". As an example, consider "*". This is conventionally used for both multiplication and to designate unresolved array dimensions.

2. No Syntax Extensions (H2).

a. Inconsistency. "The user will not be able to... define new infix operator precedences." This is in conflict with E2: "Defined types will be indistinguishable from built-in types."

b. Ambiguity, Disagreement. See our comments under C2.

c. Ambiguity. If new operators can be defined (E1) but not new precedences (H2), what are the parsing rules for these new operators?

3. Identifiers and Literals (H4).

Comment. "The language should require separate quoting of each line of a long literal." It is recommended that the appropriate connective operator (concatenation, for strings) be required between components of a multi-part literal to explicitly describe intent. This also avoids special rules about how two consecutive tokens are to be treated in certain special cases.

4. Key Words (H5).

Ambiguity, Inconsistency. "Key words will be reserved... and will not be usable... where an identifier can be used." Consider the key word TRUE. This can be used in places where a Boolean identifier can be used.

5. Comment Conventions (H7).

Comment. The following observation is admittedly sardonic, but it is somewhat amusing that the Tinman's suggested comment convention is claimed to "nearly" meet the

Tinman's requirements: if satisfying the comment requirements is difficult, what hope is there that the difficult issues can be satisfactorily addressed?

6. Unmatched Parentheses (H8).

Comment. "Failure to require proper parentheses matching makes it more difficult to write correct programs." This is incorrectly polarized: what this makes difficult is the detection of incorrect programs. We agree with the requirement, in any case.

7. Uniform Referent Notation (H9).

Disagreement. "There will be a uniform referent notation." Although we agree with the intent of this requirement, we believe that it may be exceedingly difficult to meet completely. For example, consider an array A and the assignment

```
A(I) := A(I) + 1
```

Since uniform referent applies, it should be possible to change A from an array to a function. Note that it now must be possible to have left-hand-side functions (like PL/I pseudo variables) which are user-definable. This kind of function is quite difficult to realize and results in a much more complex language. We recommend that this requirement be softened.

8. Consistency of Meaning (H10).

a. Comment. "No language defined symbols appearing in the same context should have essentially different meanings." Another wording problem: what is obviously intended is that a language symbol should not have essentially different meanings in different contexts.

b. Inconsistency. See our comments under H1.

## Section X. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic (I1).

Ambiguity, Comment. There seems to be an unfortunate entanglement of ideas here. Apparently what is intended in I1 is the statement that implementation-dependency should be avoided wherever possible. We fully concur with this objective. However, this is a totally distinct issue from defaults.

2. Compile-Time Variables (I3).

Ambiguity, Comment. If the requirement sentences of I3 were replaced by the phrase occurring later ("the programmer should be able to specify the intended object machine configuration"), the meaning of the section would be clearer. Also, referring to these specifications as "compile-time variables" is likely to cause confusion; such specifications are in fact compile-time constants.

3. Translator Restrictions (I6).

a. Comment. The requirement that restrictions such as the number of array dimensions be specified in the language definition is in conflict with D5 ("There should not be any arbitrary restrictions on the structure of data").

b. Disagreement. Specifying independent bounds on such translator parameters as the length of identifiers and the number of identifiers is overly restrictive; for example, it may be the total number of characters required by the identifiers which is the critical factor, not the maximum length or the number of identifiers.

c. Disagreement. In view of the wide variety of hardware configurations and memory capacities available within the DoD, we do not feel that reasonable bounds of the kind intended by I6 are feasible.

## Section XI. EFFICIENT OBJECT REPRESENTATION AND MACHINE DEPENDENCIES

1. Efficient Object Code (J1).

Comment. The sentence "The language should not force programs to require greater generality than they need" is hard to understand.

2. Optimizations Do Not Change Program Effect (J2).

Comment. "Any optimization performed by the translator will not change the effect of the program." The concept "change the effect" must be given a very careful definition. While the rules of algebra show that x + (y - z) is identical to (x + y) - z, ordinary floating-point hardware produces different results in some cases. For example, suppose x has a value of 0.1 and y and z have identical values of 10**20. In the first case, the result is exactly x, while in the second case, the value is exactly zero. We would argue that preservation of the algebraic effect is sufficient under "optimization", even if the numerical result is not exactly preserved. Otherwise, too many useful optimizations would be disallowed. Another such "change the effect" question is illustrated by x := 0; ...; y := z/x; with a mundane compiler, the division by zero will not be detected before run time. At run time, the appropriate exception will be signalled, corresponding recovery action (presumably) performed, and execution would continue. If a more clever compiler were to notice (at compile time) the inevitability of the division by zero, and issue an error indication and prevent execution, would this constitute a "change of effect"?

3. Machine Language Insertions (J3).

Overspecification. "The machine language insertions should be permitted only within the body of compile-time conditional statements...." The goal is correct: encapsulation of machine-dependency. However, the stated requirement seems limiting, if it precludes other suitable encapsulated forms. Procedure-level encapsulation appears particularly attractive as an alternative.

4. Object Representation Specifications (j4).

a. Ambiguity. "These descriptions [of the representation of composite data structures]... will be distinct from the logical description." The meaning of "logical description" is not clear. The intent of this requirement seems to be that the representation of a data type should be hidden in a way so that changes to the representation do not affect those parts of the program which use (rather than define) the data type.

b. Ambiguity, Disagreement. If the representation is not specified, "the object representation will be optimal as determined by the translator". First, at least some indication of the requirements for the representation must be specified (e.g., the value set of the type being defined). Second, the translator should not be required to make an optimal choice since this is exceedingly difficult.

5. Open and Closed Routine Calls (J5).

a. Comment. "An open and a closed routine of the same description will have identical semantics." This must not be construed to mean that any closed routine can be changed to an open routine. The difficulty can be seen by considering an open recursive routine.

b. Comment. "Open routine capability is especially important for machine language insertions." This is sometimes true, of course. Nonetheless, the historical preponderance of machine language coupled with high-level language has been in the area of function libraries, where speed and space considerations led to closed machine language routines.

c. Comment. "Thus, an open routine should differ from a syntax macro...." An open routine and a closed routine of the same description have identical semantics. Thus, open routines differ from syntax macros in precisely the way that closed routines do.

## Section XII. PROGRAM ENVIRONMENT

1. Operating System Not Required (K1).

Ambiguity, Inconsistency. What is meant by an operating system? Are any run-time support routines considered to be in an operating system? If so, then this requirement conflicts with G6, which requires a parallel processing capability and with B10, which requires I/O operations.

2. Program Assembly (K2).

a. Comment. The distinction drawn between separate definition and separate compilation is a good point. General practice seems to allow us to be careless with our terminology in this area.

b. Disagreement, Overspecification. It is by no means agreed that "...separate compilation increases the difficulty and expense of the interface validations...". Anyway, this is one of many implementation (not language) considerations which appear in the Tinman, but perhaps should not.

3. Assertions and Other Optional Specifications (K5).

Inconsistency. "[Assertions] will have the status of comments." This is in conflict with H7, which requires only a single comment convention.

## Section XIII. TRANSLATORS

1. No Subset Implementations (L2).

Comment. "Every translator for the language should implement the entire language." Although there can be little argument that this is a laudable goal, these Tinman requirements themselves threaten to make the goal impractical. (For example, see our comments on C1 and C4.) Paraphrasing Prof. Hoare: a prudent procedure to follow would be to strive for a language in which subsetting is not necessary, and to plan for subsets.

2. Low Cost Translation (L3).

a. Ambiguity, Disagreement. "The translator should minimize compile-time costs." Not only is this sentence disputed in two places in the paragraphs which follow it, but it does not really convey any information unless the set of constraints which must be satisfied is specified. Obviously, more time will be required during compilation to produce better object code, which sometimes will be strongly desirable.

b. Comment. "...trivial programs [should] compile and run in trivial time." While this goal seems (superficially) to be correct, it is nonetheless certain that the amount of DoD software budget spent on "trivial" programs is insignificant. If it should prove convenient, for some reason, to compile "trivial" programs ineffectively in order to make some gain in an area of high software spending, that alternative seems vastly superior. We would all like to do well in every area. But we must not be blind to the necessity of making trades to get the maximum payoff in places where it really matters. Example: programs need to be easy to read and comprehend with higher importance than they need to be easy to write.

3. Many Object Machines (L4).

Comment. "...makes the full power of an existing translator available at the cost of producing an additional code generator." Not to be overlooked are the related needs to supply other target-machine-dependent software, such as operating system interface, machine-dependent extensions and libraries, debugging aids, etc.

## Section XIV. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only (M1).

Ambiguity, Comment. "The language will be composed of features which are within the state of the art...." There is considerable disagreement in many areas as to what features are within the state of the art. Some might argue that certain of the Tinman required features are not within the state of the art (e.g., user-defined data types, uniform referents, extensive compile-time facilities). Note that even when several features are within the state of the art, a language in which they are combined may not be. In particular, research may be required on how to handle feature interactions.

2. Language Documentation Required (M3).

Overspecification. "The language will be defined as if it were the machine level language of an abstract digital computer." Although this kind of definition is certainly possible, there are other reasonable definitional approaches (e.g., axiomatic definition) that are ruled out by the requirement.

Copy No. _____

5.3.7

# DOD PROGRAM FOR SOFTWARE COMMONALITY

# HIGH ORDER LANGUAGE WORKING GROUP

## CANDIDATE LANGUAGES EVALUATION AND RECOMMENDATIONS REPORT

Prepared for

NAVAL ELECTRONICS SYSTEMS COMMAND
Washington, D.C.

Under

CONTRACT N00039-75-C-0289


COMPUTER SCIENCES CORPORATION

6565 Arlington Boulevard

Falls Church, Virginia 22046

Major Offices and Facilities Throughout the World

## PREFACE

The Department of Defense (DoD) High Order Language (HOL) Program has as a primary objective to establish a minimum number of HOLs for use throughout DoD. This volume, containing comparisons of:

| | |
|---|---|
| CMS-2 | JOVIAL J3B |
| CORAL 66 | JOVIAL J73 |
| CS-4 | PEARL |
| EUCLID | SPL/1 |

against the DoD language requirements document "Tinman," represents Computer Sciences Corporation's contribution to DoD's goal. This work was performed under contract N00039-75-C-0289 and was monitored by Mr. Robert I. Kahane (ELEX 03E) Washington, D.C.

TABLE OF CONTENTS

Preface . . . . . ii

Section 1 - Introduction . . . . . 1-1

Section 2 - Tinman Discussion . . . . . 2-1

Section 3 - Comparative Evaluations . . . . . 3-1

Section 4 - Discussion and Summary . . . . . 4-1

## SECTION I - INTRODUCTION

Like most large computer users, the Department of Defense (DoD) has been plagued with a proliferation of High Order Languages (HOL) and incompatible systems serving the "same" language. The DoD's problems are, in principle, no different from the rest of the computer user community; they are simply larger as is the use of computers. Further, DoD systems are often individually very large and very long lived. Because of the intimate integration of the computer resources with the rest of a large defense system, it has not been possible to change computer subsystems with the frequency which might characterize a commercial operation. As a result, maintenance of the computer system is both long-term and dynamic, and maintenance in many cases involves modification of the system to respond to new threats. Defense systems are often composed of interacting but independently developed subsystems, sometimes brought into existence over a period of years, all of which must be served by a common but evolving hardware base. In such an environment, the DoD finds itself spending an increasingly larger fraction of its systems resources on software. HOL commonality and the resulting flexibility would provide a powerful tool for reducing the high cost of software in DoD.

With each of the Military Departments studying the problem and making proposals for common languages, it was clear that the greatest benefit could be reaped by providing languages common across the DoD. In January of 1975, a DoD High Order Language Working Group (HOLWG) was chartered by DDR&E with representatives from the Military Departments to investigate the requirements and specifications for programming language commonality, to compare these with existing approaches, and to recommend adoption or implementation of the necessary common languages or approaches. Until the matter is resolved, the DoD will not support the further implementation of new HOL in R&D programs.

The first task of the HOLWG was to formulate a set of requirements consistent with the levies of the Military Departments. Requirements were solicited from as broad

a base as possible, to be prioritized later as required. Inquiries were not restricted to those programs presently using HOLs; rather, a major thrust of the effort was to provide HOLs to meet the requirements of those who are now constrained to use an assembly language for lack of a suitable HOL.

It has been impossible to define rigorously the exact level requirements desired; therefore, a "Strawman" of preliminary requirements was established to define this level by illustration. The "Strawman" was not intended to be complete or consistent, rather it was deliberately provocative to elicit the widest possible comment. It was forwarded to the Military Departments, and by them to their various operating agencies. In addition, it was distributed to other Government agencies, the academic community, and industry; through industry organizations and military contractors; and by direct inquiry. A number of individuals and organizations have had an opportunity to examine this document and provide inputs. The bulk of such comments were positive and useful.

The results of four months of such input were put together in a more concrete form; one which could then be representative of a fairly complete set of requirements, although still a tentative set. This document was called the "Woodenman," and it too was distributed widely. It provided a more rigorous framework for specific comments. On the basis of all inputs and the official responses from each of the Military Departments, a more complete set of requirements has evolved and is called "Tinman." This document represents a set of requirements for HOL computer programming language consistent with the input from the Military Departments.

The next task of the HOLWG was an investigation and comparison of existing HOLs to the DoD HOLWG's language requirement document, "Tinman."

A candidate set of HOLs were chosen and contracted to various industry, individual, and university groups for evaluation. The effort represented by this volume, is Computer Sciences Corporation's contribution to the task of establishing a minimum number of HOLs for use throughout DoD. Each contractor investigated and compared selected languages to the DoD HOLWGs language requirement document "Set of Criteria

and Needed Characteristics for a Common DoD High Order Language, Tinman." Specifically, each contractor performed the following tasks:

(1) For each language characteristic listed in "Tinman," the contractor will determine the "degree" of compliance of each of the candidate languages.

   (a) If the requirement is "met," the contractor will demonstrate how this is so (e.g., the presence or absence of a particular feature, computer program, etc.). Additionally, the contractor must show that this solution does not conflict with any other DoD HOL requirement.

   (b) For requirements which are "not met" or only partially met, the contractor will provide:

      1. An analysis of why the language does not fulfill or only partially fulfills the requirement.

      2. The scope of modifications that would be necessary to bring a language into compliance, and their impact on other language features.

      3. The impact of such modifications on implementations

   (c) If the requirement is such that is is not addressed in the provided language documentation (e.g., the requirement is strictly dependent on the operating system, libraries or is only a management consideration), then the contractor will so indicate.

(2) Upon completion of (1) the contractor will then identify features of the language which are not needed to satisfy the requirements with his recommendation as to whether those features should be kept or possibly eliminated.

## SECTION 2 - TINMAN DISCUSSION

COMMENTS ON

TINMAN

REQUIREMENTS

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

## Introduction

The following is a collection of recommendations for improvements in the Tinman requirements. The comments do not address questions of interpretation or obvious omissions or wording problems, except where such omissions or problems cause the intent of a Tinman requirement to be unclear.

This report has one section on the Tinman as a whole, and a number of sections on individual requirements. Not all of the requirements are commented on, many of them being so clearly reflections of currently accepted language design philosophy that no comment is needed.

The version of the Tinman requirements addressed in this report is the one dated June, 1976.

The following comments and criticisms are intended constructively. We trust that they will be received in that way.

General Comments

The principal criticism of Tinman as a whole is that it needs to be more carefully written. We say this while recognizing that it obviously has received a great deal of care and work. However, adherence to a few rules (which are, unfortunately, very laborious to follow) will result in the next version being much easier to understand. They are: (1) Be certain that all terms are clearly understood. (2) Use terms consistently. (3) Avoid using slight variations of terminology for reasons of style. The following paragraphs are short elaborations of these points. If any of the authors of Tinman find them to be so elementary as to be insulting, we apologize, but we think that the points should be made.

Assuring that all terms are clearly understood is quite difficult in a document of the nature of Tinman, because the simple expedient of defining all terms is not available. For example, it is clear that the authors of Tinman intended for the concept of type to be defined implicitly. In most cases, however, explicit definition is appropriate. In requirement P2 the word **equivalence** is used. In mathematics this word must be defined in each context, but no definition is given in Tinman and, as a result exactly what was intended in this requirement is not clear. In addition, there are a number of computer terms which the authors apparently thought would be clear to any professionals in the field, but which are open to interpretation (e.g., **conformable arrays** in requirement B7 and free union in requirement E7).

The inconsistent use of terms is universally recognized as something to be avoided, but it is very difficult to do without some sort of mechanical assist, such as a kwic index. Even with such an assist the job is tiresome. In requirement D5 it appears that the word **range** is being used in the sense of "set of", a meaning different from that used elsewhere in Tinman. The result is uncertainty as to the exact intent of this requirement.

## Requirement A2

We maintain that a bit string type is necessary (see the comments on B10).

It is not clear from the statement of this requirement if literally a character type is being called for or if character type is an ellipsis for character string type. Given a character type and the definitional mechanisms called for in the Tinman, the common character string capabilities can be achieved. We believe, however, that the character string type is useful enough to warrant its inclusion as a compiler-defined type. Full-blown variable length strings are probably not needed. For most applications variable length strings with maximum lengths known at compile time suffice. Indeed, for most application strings with fixed length (fixed at compile time) are probably adequate.

We are confused by the mention of some type generators in both requirements A2 and E6. If this was intentional, then we have missed the point. It appears that requirement A2 should call for the "basic" types (integer, real, Boolean, character, character string, and bit string) and requirement E6 should cover the type generators (records, arrays, enumeration types, and powersets).

## Requirement A4

To treat fixed point numbers as exact quantities often results in a great deal of run-time overhead. Usually multiple precision integer arithmetic is involved. If the fixed point capability is elaborate enough, full-blown multiple precision rational arithmetic could be necessary. Based on CSC's experience with Navy embedded computer systems (for example, NTDS), it is doubtful that the user community would accept such overhead.

Based on this same experience, we think that something slightly different is intended: Whenever two fixed point quantities are added or subtracted, any shifting necessary to line up radix points must be left shifts. That is, the usual floating point discipline of a right shift of one operand, discarding bits which correspond to "invisible" error bits of the other operand, is unacceptable; the other operand is considered not to have any "invisible" error bits -- it is exact. CMS-2 compilers perform their fixed point additions and subtractions in this manner for precisely this reason, and at the strong insistence of their user community.

## Requirement A5

It is not clear if this requirement is for an ability to specify character sets or collating sequences -- two quite distinct concepts.

Both of these concepts are also distinct from that of an enumeration type, so to be able to specify either using the notation of enumeration types is a conflict with requirement H10.

The remainder of our comments are based on the assumption that the subject is character sets.

In our view, the character set used is a property of the internal representation of the value, not of the value itself or of the operations available for it. Hence a character set is not a type, and which set is used depends on the containing variable (see also the comments on B1). If two character or character string variables have different representation -- different character sets -- then implicit, automatic conversion should occur when the value of one is copied to the other. Conversion might not always be possible, because a character may appear in only one of the character sets, but the same is true for numeric variables having different ranges, and the Tinman explicitly prohibits ranges from determining types (requirement D4). Explicit character set conversion operators are probably needed also, especially for literal values.

## Requirement A6

Why must the lower subscript bound be determinable at compile time? If the user really needs a dynamic lower bound, this rule must make his code more awkward, harder to read, and a bit harder to optimize.

The statement that the default lower bound (our interpretation of preferable lower bound) is zero is arguable, particularly when one of the justifications is that it contributes to clarity. The great majority of applications of arrays are "natural" ones -- those which have a concept of a first element, a second element, etc., and a last element, and these should be represented by 1, 2, ..., and the array size, not 0, 1, and one less than the array size. Arrays which have a lower subscript bound different from 1 arise, for the most part, from specialized mathematical formulas, which probably play a very minor role in embedded computer applications in the Department of Defense.

Even the requirement that the values be contiguous might be unduly harsh. It is simple to implement non-contiguous subscript values without the expense of garbage collection; the compiler can simply construct a mapping function, which a programmer who needs such a capability would otherwise have to do himself. Requirement J1 prohibits this general case from affecting the efficiency of the ordinary, contiguous subscript, case. The only question, of course, is whether there is enough need to justify the inclusion of such a feature.

## Requirement B2

The text of this requirement contains the sentence: "For floating point numbers identity will be defined as the same within the specified (minimum) precision." This cannot be done efficiently on binary machines, assuming that the precision specification is stated in decimal. Furthermore, ignoring any questions of efficiency, to build such a predicate into a language is dangerous because it would tend to encourage programmers to gloss over the difficulties of working with floating point data. The initial accuracy of floating point data is only one consideration when a comparison for equality is performed, and it may not even be the most important. The build-up of error in intermediate calculations must be taken into account, usually in an ad hoc fashion in each use.

## Requirement B5

Avoiding truncation of the most significant digits implies run time checks. We recommend that the programmer be able to avoid this overhead (see also the comments on F7).

## Requirement B6

This requirement specifies that "and" and "or" on scalars will be evaluated in short circuit mode. The intent of the qualification on scalars is unclear.

In addition, specification of short circuit mode conflicts directly with Requirement C1. We believe that short circuit mode evaluation, if provided, requires special syntax. It is hard to see how the given example improves clarity and maintainability. Dependencies should be clearly shown; for example:

```
if I < 7 then A(I) > X else false
```

or (using our own syntax)

```
if I < 7 (and A(I) > X).
```

A maintenance programmer cannot miss the intent here, as he easily can if a simple Boolean expression is used.

Hence we recommend that the second sentence of the requirement be replaced by:

> Short circuit mode evaluation will be guaranteed only if special syntax, clearly showing the dependencies, is used. Otherwise, the programmer

will not be able to count on short circuit mode evaluation, or to assume that it will not occur.

The last clause of course means that the programmer can not depend on every function in an expression actually being called. If a function must be called, it can be put into a separate statement.

## Requirement B8

This requirement does not explicitly forbid "mixed-mode" arithmetic, although that seems to be in the spirit of the requirement. "Mixed-mode" can be defined without resorting to implicit conversions by defining the arithmetic operators to depend on the type of their operands as well as their values. Of course, everyone knows that an implicit conversion is performed prior to execution of the operation, but if the conversion is done by the firmware it becomes difficult to argue that anything shady is going on. If a prohibition is intended, it should be explicitly stated.

With regard to the question of whether or not "mixed-mode" arithmetic should be prohibited, it becomes difficult to argue against when no information is lost during the conversion; for example, when an integer value is converted to fixed point or floating point, a fixed point value is converted to floating point, or a value of some type representing a real number is converted to the equivalent complex number. Indeed, many language which have fairly strict type-checking allow something like this for constants at least through the device of specifying numeric constants to be typeless, specifying only a value -- not a type. It is probably best to specify that those conversions which do not lose any information (spelled out in detail, of course) can be performed implicitly.

If multiple character sets are to be supported, as indicated by requirement A5, the question of implicit conversion between character sets should be addressed. Again, such conversions should probably be permitted in any case in which no loss of information can occur.

Many programmers find conversions between integer and Boolean and between enumeration types and integers useful. We recommend that they be included.

## Requirement B10

We recommend that something analogous to the FORTRAN FORMAT-controlled conversion also be required, both for I/O and internal conversions. In addition, we recommend that it be possible to copy a binary bit string into a variable of any type and vice versa. This too would be provided not only for I/O -- that is, it should be possible to

move a value of type bit string into any variable, without type conversion, and vice versa. This is sometimes the only efficient way to move data between machines, and may be the only way to break apart values whose structure and type is dynamically determined. It is, of course, machine and representation dependent, and allows escape from type checking. Requirement J3 states that machine dependence shall be possible. Type consistency rules, in our view, should not be regarded as a system of laws that the programmer cannot violate. Instead, they are a way of using redundancy to ensure that the programmer means what he writes. However, the programmer knows more than the compiler, and may not always be able to stay within the constraints laid down -- he must be allowed to violate them, if he states clearly that such is his intent. This is analogous to providing structured programming control structures, but keeping the **goto** for use where necessary (see the comments on C2).

## Requirement C1

To write two function references, with both of the functions having side effects, and then expect the side effects to occur in a specific order is a terrible programming practice. Yet the effect of this requirement will be to give it an aura of legitimacy.

Also, in practice, this rule restricts optimization quite a bit. Furthermore, the meaning is not clear for some cases (e.g., embedded assignment). A better requirement would be:

> "There shall be no rule in the language specifying that every argument of an expression is to be evaluated when the expression is evaluated, nor making the value of an expression dependent on the order of evaluation of its arguments, except where such dependency is clearly specified as part of the definition of an operator."

This rule can even be enforced by the compiler, in one sense, by having it generate "side-effect prone" operands (function references, embedded assignments, etc.) in a random order, using as its randomizer some datum over which the programmer has no control (e.g., some low-order bit of the hardware clock), which would presumably bring out any incorrect dependency on the order of occurrence of side effects early in the debugging stage. We know of no compiler which has used such a technique.

## Requirement C2

"Few levels" should be more precisely defined. One level, as in APL, seems plainly to few. Wirth now believes that even Pascal has too few levels.

## Requirement D2

The requirements spelled out here are good, in spite of any possible disagreements from compiler writers. Another one needs to be added: Alternative literal forms for the same quantity will have the same value. For example (using FORTRAN notation), a comparison of 1.1 and 0.11E1 should always give an equal result. Some compiler writers might well complain about this too.

## Requirement D3

Some comment should be made about the initialization of variables shared between independently compiled programs -- e.g., in COMMON. We believe that for any named block of such variables, initialization should be specified in no more than one place, or that identical initialization must be specified in all declarations (to permit copying of declarations).

## Requirement D6

Our comments on this requirement are contained in the paper "A Note on 'Pointers'" by Christopher Earnest (September 1, 1976), presented at the Cornell conference.

## Requirement E1

The need to be able to define new data types is clear. However, at the risk of being thought unfashionable, we maintain that new data operations should not be definable in the language, except by means of procedures and functions. In other words, new infix operators are neither necessary nor desirable, for the following reasons:

(1) H2 precludes the definition of new precedence levels, and we agree -- too many levels is confusing. But this means that a new infix operator must be at the same precedence level as an existing operator, which can also lead to distortion, and which means that new operators are clearly different from built-in ones. The problem goes away if new infix operators are not allowed; functional notation specifies precedence clearly.

(2) If new symbols are introduced as operators, then very soon the APL problem of overpunching, etc., arises -- that is, the primitive alphabet becomes too large for ready comprehension. If words are used as new operators, perhaps enclosed in special brackets, then what is the advantage over function notation?

(3) The requirement for component by component operations (R7) means that the most natural way of introducing new operators -- as new meanings for existing operators -- is often not possible. Matrix multiplication, for example, must be expressed by means of a new operator (as it is in APL). Again the advantage over functional notation is unclear.

(4) Too many operators makes a language very unreadable, in our view and in Dijkstra's view also. He has strongly criticized APL for exactly this. More generally, he emphasizes the need to restrict, rather than to expand, the basic tools we work with, and we concur wholeheartedly. Addition of new operators to a language should not be easy; making it so invites abuse.

More philosophically, we would argue that a language is best extended through its literature (i.e., for a programming language, through its programs), in which existing basis elements are combined in new ways. This is certainly true of natural languages: New words are introduced very rarely, and new meanings for existing verbs (the operators of the language) almost as rarely.

We believe that the set of operators which deserve to be generic grows slowly. The basic set of operators should probably include more than the Tinman requires -- for example, matrix operations and complex arithmetic -- and subsets of the operators should be defined to be supported by different level compilers. Then before a new operator is introduced, the need should be clearly demonstrated, and if warranted, the compilers and the language documentation would be modified to support it. Such additions should happen rarely, and implementation through the compiler often leads to greater efficiency. If someone does not understand enough about the structure of a language to understand its compiler, should he in fact be able to modify the language?

These comments do not apply to new types or new data structures. These can be thought of as adjectives applied to a noun to restrict its scope, or as strings of nouns, embodying additional attributes. In other words, the language must be able to deal with new kinds of objects. The above comments do apply to the introduction of new ways of defining data structures, and here we and the Tinman seem to agree.

It is of course necessary to be able to do **something** with new types, and new data structures. The only way to introduce new scalar types is by enumeration, and all enumeration types share a common set of operations. The interesting cases are new structured types. The structure-defining mechanisms must include generic specification of the way in which individual components of a structure can be referenced and modified; for example, all array types have the same access conventions, whether their members are real, integer, or themselves structured. It is also possible to define generic rules for the meaning of relational operators with a new ordered type; if these are not sufficient, functions can be used. In some cases, reference to or modification of a variable should have side effects -- for both built-in and new types.

The Tinman appears to partially share these views -- Boolean could be provided as an enumeration type, with sufficient definition of new operators. The fact that it is specified explicitly as a built-in type seems to indicate a realization that new operators should affect the base language.

Requirement E4

This requirement conflicts directly with requirement B7. Independently of that, we disagree with the requirement for reasons explained at length in the comments on requirement E1.

Requirement E7

We recommend that the following requirement be added: "It must be possible to compile a program so that there is no overhead caused by run time type checking." This is intended as a language requirement. Discriminated union implies run time type checking, supplied by either the compiler or the programmer (as in ALGOL 68). For a fully debugged program, the overhead of this must be avoidable. If the language insists on, for example, a tag field for a discriminated union, then the compiler cannot remove it even if run time checking is not wanted. There are many ways in which a program can determine the type in a given instance; a tag field should not be mandatory.

Requirement F2

This requirement, together with requirement E5, implies strongly that the Tinman intends a type and its operations to be defined in an isolated "cluster", with the objects of the type inaccessible outside the cluster except using the defined operations. This does not provide for new operations which have arguments of more than one type, and which must have access to the representation of the data objects. For example, if a new set type is defined for sets built at run time, an "add member" operation might need to access both a (normally inaccessible) component of an element and the control information for the set.

Better than a "cluster" for each type definition is a "cluster" for a collection of type definition. the Tinman should make this the requirement.

Requirement F6

While the meaning of the term compool is less restricted than it once was, it has remained true that a compool contains only things useful at compile time. Moreover, a library might contain a compool, but why must a compool be able to contain a library? These distinctions seem useful. We don't see the advantage of merging two distinct concepts into one.

## Requirement G2

The Tinman's attitude toward the goto seems ambivalent. It is not eliminated entirely, and there is no restriction against using it to exit from an entire nest of loops (see G4). (Our reasoning is as follows: Tinman does not explicitly require any control structure to be a scope. The use of a **goto** is restricted to its most local scope. If control structures are scopes, then the **goto** is thereby restricted to the point of uselessness or, worse, to being used poorly only. Therefore, Tinman indeed did not intend for control structures to be scopes.) Yet there is a restriction against using a **goto** out of an allocation or procedure scope. As requirement G7 recognizes, such exits are clearly necessary; special mechanisms are therefore required. Exception handling parameters (requirement C7) are required as one such mechanism; no mechanism is identified, or even explicitly mentioned, for exit from allocation scopes.

Note that any implicit actions (e.g., release of storage, stack popping) that would be connected with a **goto** out of an allocation or procedure scope are necessary in any case for the special exit mechanisms. Less complex, but similar, mechanisms are needed for exit from a nest of loops. If the use of a **goto** is thought to be confusing or error prone in the first case, why not in the second also? That is, if special exit mechanisms are required, why not spell out exactly what is wanted where, and eliminate the **goto** entirely?

In fact, our recommendation would be to keep the **goto**, and allow it to be used in an almost unrestricted manner (a prohibition against using a **goto** to enter a control structure or an allocation scope is certainly reasonable), but provide enough other tools so that its use was rarely necessary.

## Requirement G3

The use of the complementary clause of a control structure (**else** after **if then** or **otherwise** after **case**) with a null statement list occasionally makes a source program easier to read. This usually occurs when the set of statements following an **if then** or the partitioning being specified by a **case** is complicated, for example. In such cases the complementary clause has the visual effect of a closing bracket for the control structure and tends to convince the reader that he has not

missed anything. It seems a bit severe for the language to require the complementary clause, however. For example

```
IF X < 0 THEN X = X + 1;
```

would simply become cluttered by the inclusion of an **else** clause. Such questions should probably be left to individual style. (Even in tightly controlled developments programmers should be allowed some freedom.)

Requirement G4

We applaud the requirement that a terminating predicated be allowed to appear anywhere in the loop. As noted above, this also strengthens our **goto** argument.

Requirement G5

We concur wholeheartedly with the decision to make recursive procedures a requirement. We believe that the costs of recursion are justified by its contribution to clarity and conceptual simplicity for a user.

However, we do not agree with the restriction against nesting recursive procedures. We do agree that such nesting occurs rarely -- which means the display has only one entry normally anyway, if the compiler knows which procedures are recursive. The restriction therefore has no real run time advantages, and we would prefer a requirement calling for declaration of the fact that a procedure is recursive. In general, a compiler cannot develop precise information for itself (e.g., the fact that R calls P and P calls R does not necessarily mean that either is recursive). With or without the restriction against nesting, the declaration is required to avoid extra cost.

If the restriction against nesting of recursive procedures is nonetheless kept, then we note that it could be relaxed somewhat. It is only **recursive** procedures in recursive procedures which cause a multi-entry display. Other uses of stack storage (e.g., **loc**) may cause additional pointers, but the implementation choices are the same here with or without recursion. It is a bit tricky to avoid extra costs due to recursion when calling (non-recursive) formal parameter procedures, but it can be done.

Requirement G8

As the Tinman correctly remarks, synchronization of parallel processes can be done through exclusive access to data, required by G6. It can also be done by calling operating system or other synchronizing routines. Hence, the meaning of this requirement is unclear -- must additional synchronization tools be provided, or are the two mentioned enough? We would opt for the second alternative; additional tools should not be outlawed, but neither are they necessary, and deciding on a set compatible with various different environments is very hard. Note that the language BLISS, which supports asynchronous processing, provides no synchronizing primitives, presumably for these reasons.

The meaning of assigning priorities is also not clear. Typically, relative priorities change with time, to preclude endless waits for low priority processes, or to make it possible to meet deadlines. We certainly oppose building a comprehensive scheduling mechanism into the language, because different operating system disciplines would render much of it meaningless. Note that scheduling can be programmed in any way desired, by delaying processes to wait for access to shared data. Again, it is not clear why the additional requirement is necessary, and what it accomplishes. In a particular environment, or for particular uses (e.g., simulation), scheduling and specific synchronization mechanisms can of course be programmed, but requiring general ones seem untenable. The problems are illustrated by CS-4, which has nine scheduling procedures, four synchronizing ones, and six parameters describing priority! It is not clear that these are meaningful in all cases, or even that they are sufficient.

Treatment of hardware and other interrupts is a different question. Here it would be possible to provide tools in the language, as PL/I has done, for example. However, these add quite a bit of mechanism to the language. It should be possible to tell what statement caused the interrupt (where applicable). It must be possible to enable and disable interrupts, to specify the action to be taken on interrupt as a special kind of procedure or process, to cause control to continue where interrupted (and exact definition of this is extremely hard) or to exit form the interrupt procedure without resuming. Debugging of interrupt procedures presents special problems (in PL/I, SIGNAL and CONDITION can be used, but this in not quite the same as an actual interrupt). Moreover, control over interrupts has many of the undesirable traits of the goto; control in interrupted in a definitely "non-structured" way. Therefore, we question the requirement -- the added complexity may not be worth the gain.

In summary, we agree only with the part of the requirement calling for access to real time clocks, and for delays based on elapsed time or the occurrence of a specified time. The other facilities called for should certainly not be outlawed, but neither should they be required of the language.

Requirement H4

We suggest the addition of a requirement that every literal be self-identifying as to type. To prevent confusion, and make parsing easier, context should not be necessary to determine the type of a literal. The requirement does not preclude the use of immediate qualifiers, as in for example:

complex: (1.3,4.7)

Requirement I4

The requirement of conditional compilation is a good one, but we believe strongly that statements which are to be executed at compile time should be clearly identified as such. Otherwise, the flow analysis problems for the translator can be severe, and there is no reason to impose this extra burden.

Requirement J2

It is difficult to understand this requirement as a language requirement. Any optimization will change the effect of a program in some sense. Any so-called optimization which changes the semantics of a program is actually a bug in the compiler. If this requirement needs to be stated, it should be moved to section L or M.

Requirement J3

A useful feature would be to require the encapsulating statements to contain information specifying the hardware features for which the dependent code is being written. When a program written for one object machine is compiled for another, the compiler would be able to flag machine dependent sections which have not been modified. (Notice that including machine dependent text within compile-time conditional segments which depend on some compile-time variable set by the programmer merely helps generate an error-free source listing. It does not point out sections of the program which must be modified.) For assembly language segments the specifying information could be as simple as the symbolic name of the object machine. The identifiers for the symbolic names could be administered by the language controller, to ensure uniformity of implementations, and would not have to be treated as reserved words.

Requirement J4

Requirement J4

As we suggested under requirement J3, the hardware features assumed should be specified in the encapsulating statements, but here the specifications could often be more general than simply the object machine. For example, the number of bits per word is a feature which could span a number of machines.

# A COMPARISON OF
# CMS-2
# to
# TINMAN

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

## Introduction

This report gives a comparison of the language CMS-2 to the Tinman language requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1976, Section IV). For the purposes of this comparison, CMS-2 is considered to be defined by:

Users Reference Manual for Compiler, Monitor System-2 (CMS-2) for Use with the AN/UYK-7 Computer
M-5035, Vol. 1 and 2
FCDSSA - San Diego, Cal.
15 August 1975

CMS-2Y Programmer's Reference Manual (Preliminary Version)
M-5049
FCDSSA - San Diego, Cal.
1 October 1976

Tinman contains 78 language requirements. This report compares CMS-2 to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used. (Occasionally no text is needed if a requirement is totally satisfied.) If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - Only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

(The symbols P, P+, and P- will often be used with requirements which are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

The report concludes with two summaries. The first is of the features of CMS-2 which are extraneous to Tinman and the desirability of retaining each of them. The second is of the language as a whole and the desirability of modifying it to bring in into line with the Tinman requirements.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs. ....P

The language permits the user to explicitly specify the type of each variable and component of a composite data structure. The entity and its type may be declared with a variable or field (component of a record) declaration statement. If the type is not specified in the declaration the entity is assigned the type specified by a mode declaration, or, if there is no mode declaration, a default type. In addition, an optional declaration (OPTIONS MODEVRBL) allows the user to use local variables without being required to declare them. The type assigned to them is the type specified by the mode declaration or default. In addition a table may have a major index which dynamically controls the number of actual items in the table. Specification of a major index is considered a variable with signed integer typing of 16 bits length.

Since the mode declaration and MODEVRBL option are error prone (even though the compiler "glaringly" flags all MODEVRBL data units) these options are seldom used. It is a matter of policy for users to declare all data units.

Removal of the MODEVRBL option so that all entities must be declared and requiring a major index to be pre-defined would be simple modifications to the existing implementation.

Type checking is performed at compile time. However, the full intent of TINMAN is not met. All numeric data units are considered to be the same type for assignment and expression evaluation and implicitly converted if necessary. (See comments under A2 and B8.) Data units may share the same storage (OVERLAY feature) creating a free union, but discrimination is not permitted. (See comments under A7.)

---

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators. ....T

Integer .....T
Floating Point .....T

Fixed Point .....T
Boolean .....T
Character String .....T
Arrays .....T
Records .....T

All of the above are built-in types in CMS-2. For type checking purposes all numeric data units are considered to be one type.

CMS-2 has a data type called a TABLE which consists of any number of items. An item is an arrangement of atomic data units, i.e., a record. A TABLE may be an array of items (records) or a single item (record) but a record may not contain an array or a record. To permit records to contain arrays or other records would require a major change to the language and its implementation.

In addition to the binary valued Boolean (true, false), the language contains a multi-valued Boolean or bit-string. Only the binary valued Boolean may be declared. But any data unit can be considered to be a bit-string and its bits may be accessed by the BIT modifier or manipulated by the Boolean operators AND, OR, COMP (complement) and XOR.

CMS-2 also includes an enumeration type, the status variable.

---

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code. ....F

Global arithmetic precision specification mandatory .....F
Individual variable precision specification permitted .....F

The language does not permit a precision to be specified for floating point arithmetic or for individual floating point data units.

CMS-2 is a language that compiles for a single machine, the AN/UYK-7. Floating point data units correspond to the AN/UYK-7 doubleword format for floating point numbers. They always have a precision of 31 bits.

---

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler. ....P+

Treated as exact quantities .....T
Range and step size determined at compile time .....P+
Scaling handled automatically .....T

Fixed point numbers are treated as exact quantities. The minimum number of bits needed to represent the number, and the number of fractional bits are specified by the user in the declaration. This determines the maximum absolute value the number can assume. The minimum absolute value cannot be specified. Scale factor management is handled by the compiler unless overridden by the scaling specifier appended by the user to arithmetic expressions.

The language has a DEBUG option which permits the user to specify a complete range for each data unit. Instructions are generated to provide dynamic range checking. Since the mechanism is available to accept a complete range specification it would be a relatively simple modification to permit a complete range to be specified in the data unit declaration statement. However, instruction generation for dynamic range checking should remain optional.

---

A5. Character sets will be treated as any other enumeration type. ....P-

New sets can be defined as enumeration types .....F
ASCII and EBCDIC are provided .....P
(Conversion capability between sets is available) .....N/A

The implementation of the language uses the 64 character ASCII subset. No other character sets are permitted. The compiler assumes that all peripheral devices use the same character set.

To allow the user to specify the program literal and order of characters would require only a translation table for conversion from the internal representation and collating sequence used by the compiler to the character set and collating sequence specified by the user. Translation would only be necessary by the portion of the compiler that generates character constants, and for input/output with peripheral devices that use a character set different from that defined by the user.

If the character set defined by the user can have characters that are not in the 64 characters ASCII subset the character set recognizable by the compiler must be expanded.

---

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope. ....P

Number of dimensions is fixed at compile time .....T
Type is fixed at compile time .....T
Lower subscript bound is fixed at compile time .....T
Upper subscript bound is fixed at scope entry .....P
Subscripts only integers or from an enumeration type .....P
Subscripts will be from a contiguous range .....T

The user must specify the number and size of each dimension. The subscript values can only be integers and must be contiguous values starting at zero. The maximum size of an array is fixed at compile time, except for simple tables (one-dimensional arrays) whose maximum size can be fixed at load time. In addition, the size of simple tables can be dynamic by declaring a major index, a variable that controls the upper bound on the table subscript. Declaration of a major index automatically defines the major index. Its type is a signed integer of 16 bits, and its scope is the same as the scope of the table.

Subscript values can be values of an enumeration type data unit only if the enumeration type data unit is explicitly converted to an integer by use of the BIT operator.

A lower bound on the subscript range other than zero or one would require a change to the array declaration syntax and a corresponding change in the symbol table.

Extension of the major index feature to all arrays would be a modest change to the compiler.

---

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time. ....P

Alternative structures for records are possible .....T
Discrimination condition may be any Boolean expression .....F

Records (items of a table) may have alternative structures either by user packing of the fields so that they share the same storage, or by use of the field overlay feature for compiler packed fields. The structures are fixed at compile time but cannot be discriminated at run time.

The scope of modifications necessary to implement run time discrimination for alternative structures is dependent on the syntax and semantics to be used.

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value. ....T

Automatically defined for any type (except...) .....T
Available for individual components .....T
(Assignment and reference via functions) .....F

Assignment and reference is defined for all data types including arrays and records and for their components. In addition, word components of arrays and records can also be assigned and referenced.

---

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity. ....P+

Any two atomic data units of the same type (all numeric data units are considered to be the same "type") may be compared for equivalence using the identity predicates EQ (equals) or NOT (not equals). Arrays may not be compared. Components of arrays (fields) may be compared if they are of the same type.

The modification of the existing implementation to permit comparison of conformable arrays would be minor.

---

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration. ....P+

Built-in for all numeric and enumeration types .....T
Ordering can be inhibited when desired .....F

All six relational operations (EQ, NOT, GT, GTEQ, LT, LTEQ) are built-in for all numeric and enumeration types. It is not possible to inhibit the ordering of the values for an enumeration type.

An extra piece of syntax would be required to indicate an unordered enumeration type. The modification necessary would be trivial.

---

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation. ....T

Addition .....T
Subtraction .....T
Multiplication .....T
Division with real result .....T
Exponentiation .....T
Integer and fixed point division with remainder .....T
Negation .....T

Arithmetic operations use familiar notation: addition, +; subtraction, -; multiplication, *; division, /; exponentiation, **. Negation uses the unary operator -. A division with real (floating point) result occurs if either of the operands is real. The remainder of any fixed point division may be obtained by using the remainder operator (SAVING <data unit>).

---

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers. ....P

Never from the left for data within range .....T
Never on the right for integer and fixed point .....F
Implicit floating point rounding beyond precision allowed .....P
(Run time checks can be avoided) .....T

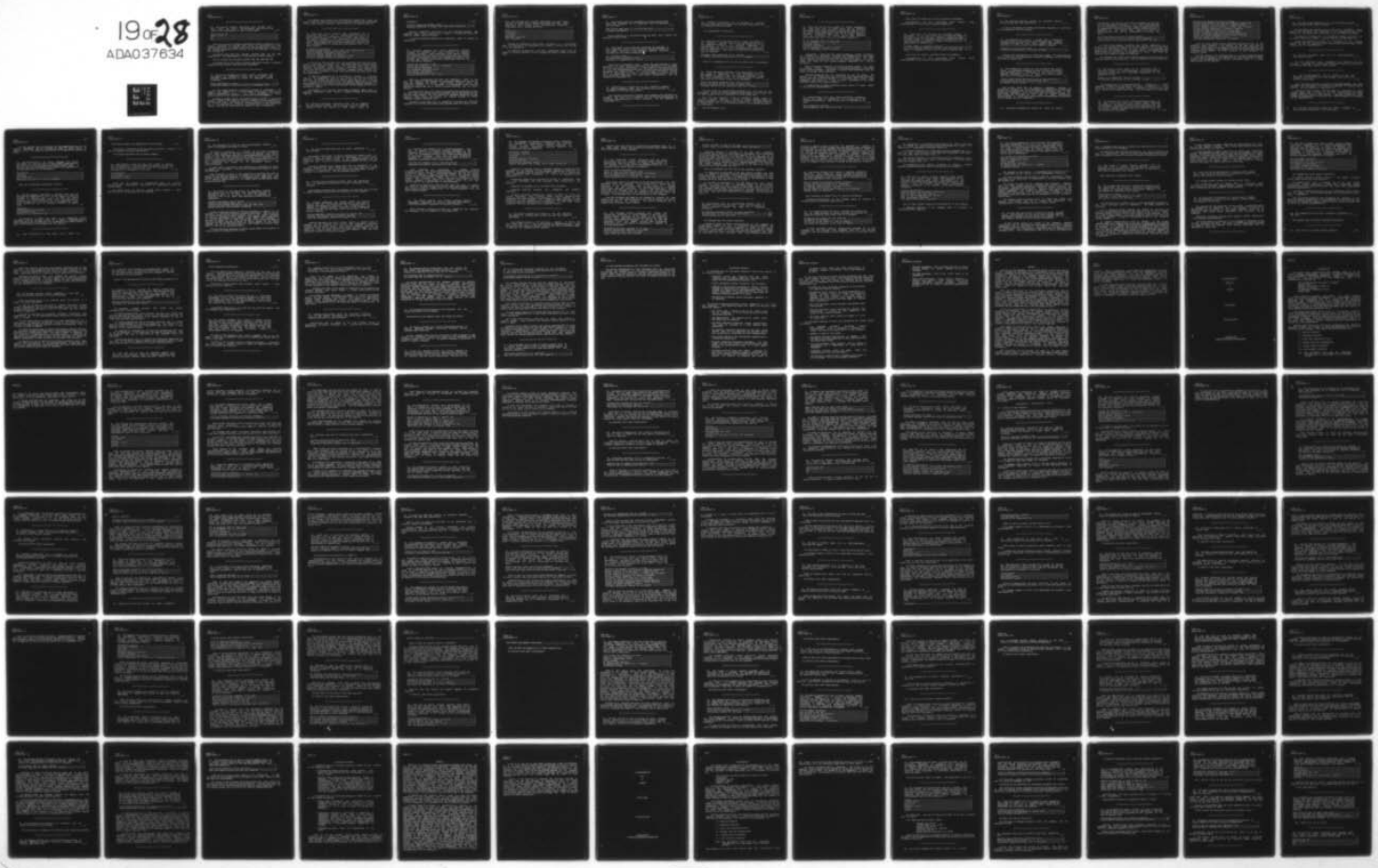
For arithmetic and assignment operations, CMS-2 never truncates the most significant digits if the data are within range. Implicit truncation is performed on the right and a warning message is issued. For floating point data units the user can specify in the data declaration whether implicit rounding is to be performed or no rounding is to be performed.

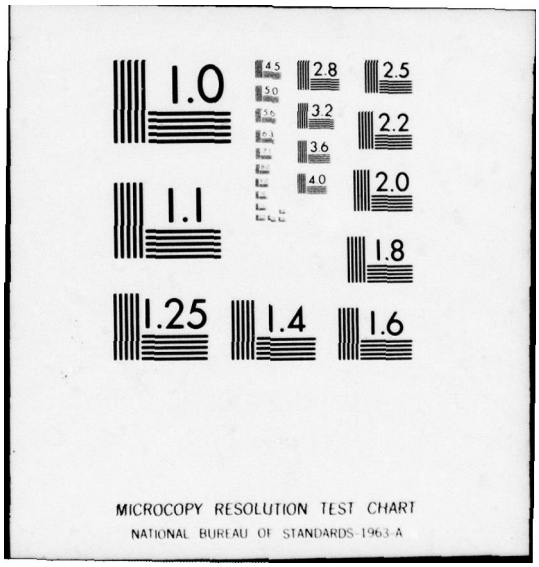
Run-time checks for truncation could be added at modest cost. However, run-time checks are costly to the user.

AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

19 OF 28
ADA037634




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

---

B6. The built-in Boolean operations will include "and", "or", "not", and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode. ....P+

Short-circuit and .....T
Short-circuit or .....T
Not .....T
Xor .....P

CMS-2 has two sets of Boolean operations: Boolean connectors (i.e., logical connectors on Boolean data units or expressions as in SET <Boolean data unit> TO <Boolean data unit> AND <Boolean data unit> $); and Boolean operations on bit strings (as in SET <integer> TO <integer> AND <integer> $).

For Boolean data units the language provides AND, OR and COMP (complement). The implementation of AND and OR is short circuit.

For bit strings the language provides AND, OR, COMP and XOR.

Implementation of the logical operator XOR would require a moderate addition to the logical code generation sequences.

---

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure. ....P

Scalar operations on arrays .....P-
Assignment between records and arrays of conformable type .....T

The only scalar operation permitted on arrays is assignment. For example, SET TABARY TO 5 $ where TABARY is an array. Each word of TABARY is assigned the value 5. All scalar assignments to tables (arrays) and to items (records) of tables are word assignments.

Assignment between tables and between items of tables is permitted without regard to the types or logical arrangement of the components of the tables or items. The tables or items do not need to be conformable, nor of the same length. Transfer is accomplished on a word basis, the length of the shorter determines the number of words transferred.

To require type checking and conformability checking on array and record transfers and to allow logical assignment between non-conformable arrays would require a moderate modification of the current compiler.

---

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors. ....P

No implicit conversions .....F
Explicit between integer, fixed point, and floating point .....F
Explicit between fixed point scale factors .....T
(Explicit between integer and Boolean) .....P
(Explicit between integer and enumerated types) .....P
(Explicit between different enumerated types) .....F

In CMS-2 all numeric data units are considered to be the same type. During expression evaluation and assignment implicit conversions are performed on integer, fixed point and floating point data units. Explicit conversions between integer, fixed point, and floating point data units are not required nor permitted. However, scale factors can be explicitly specified on fixed point data units and fixed point data unit expressions.

An explicit conversion can be performed on a Boolean, a character type or an enumeration type to an integer by the use of the BIT operator. An integer (or any other numeric type) may not be converted to a Boolean or an enumeration type. Any numeric data unit or any data unit converted to a bit string by the BIT operator, may be converted to a character type by use of the CHAR operator (if the number of bits is a multiple of eight).

Replacement of the implicit conversions on numeric data units by explicit conversions would require a moderate change to the existing compiler.

---

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is

truncated. ....P+

Implicit conversion between ranges .....T
Exception condition on integer and fixed point truncation .....P

Explicit conversion operations are not required between two numerical data units that are out of range. If truncation occurs the compiler issues a warning message.

A run-time excention condition for truncation could be added at modest cost.

---

B10. The base language will provide operations allowing programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent. ....P+

Sending and receiving of data .....T
Sending and receiving of control information .....T
Dynamic device assignment .....F
User exception condition control .....P
Installation independence .....T
(Data formatting capability) .....T
(Reading and writing of bit strings) .....T

CMS-2 has a rudimentary high level I/O capability. With it the user can transfer data between his program and named files on tape or disk, and standard I/O devices (card reader, printer, punch, and operator's console), in a formatted or unformatted form. The I/O operations require the use of library run-time routines and the CMS-2 monitor.

The CMS-2 high level I/O capability provides the equivalent of what one would need in a typical batch environment, and is not sufficient for tactical data systems. The I/O functions in embedded executives in most tactical data system applications are usually written in direct (embedded assembly language) code.

Expansion of the high level I/O capability to provide the functions necessary for tactical data system applications would be a major effort.

---

B11. The language will provide operations on data types defined as power sets of enumeration types (see F6). These operations will include union, intersection, difference, complement, and an element predicate. ....F

Union .....F
Intersection .....F
Difference .....F
Complement .....F
Membership predicate .....F
(Set inclusion) .....F

Although an enumeration type (status variable) is a permissible type the powerset of an enumeration type cannot be defined, nor can any of the set operations.

The inclusion of powersets and their operations would have no impact on existing features in the language and could be added at modest cost.

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right. ....T

Side effects must occur in left-to-right order .....T
(Embedded assignments) .....F

Those arguments of an expression that can cause side effects are evaluated left-to-right.

---

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized. ....P+

Few precedence levels .....T
No user-defined precedence levels .....T
Operands of an operation are obvious .....P+

CMS-2 has six precedence levels. Within the same precedence level evaluation is from left-to-right unless the operation to be performed first is specified by parentheses or a unary minus is encountered before or after an exponentiation operation, in which case the operations are performed from right-to-left. The left-to-right evaluation defines X/Y/Z as equivalent to (X/Y)/Z. The only exception to these rules is the expression evaluation for substitution declarations. See the comments on requirement H10.

---

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed. ....P+

CMS-2 permits expressions anywhere both constants and variables are allowed. The form of an expression for substitution declarations is different from expressions in other contexts. See the comments on requirement H10.

---

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time. ....T

This requirement is fully met.

---

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to ease of learning. ....T

Parameter rules consistent in all contexts .....T
No special operations applicable only to parameters .....T

There is a consistent set of rules for each class of parameters.

---

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter. ....P

Actual and formal parameters will agree in type .....P+
Rank of parameter arrays is fixed at compile time .....T
Parameter array size and subscript range can be passed .....P

The type rules for actual/formal parameters are the same as for assignment: Actual and formal parameters must agree in type except that all numeric data units are considered to be the same type.

The size and subscript range of parameter arrays cannot be explicitly passed. However, a formal or actual procedure input (or output) parameter can be a table with a major index. The major index on a simple table, i.e., a one dimensional array, can be passed explicitly as a parameter.

See the comments on A6.

---

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters. ....?

Act as constants (call by value plus) .....F
Act as variables (call by reference) .....P-
Exception control .....T
Procedure parameters .....F
(Act as variables, but call by value) .....T
(Act as variables, result parameter) .....T

A procedure in CMS-2 may have input parameters, output parameters and abnormal exit parameters. Abnormal exit parameters are label names on the calling side and condition names on the called side. The label names need not be within the scope of the calling procedure but must be within the scope of the system-procedure containing the calling procedure.

Input and output parameters are called by value (except for input of indirect tables). Assignment to the formal parameter is permitted within called procedure. Assignment to formal input parameters does not affect the actual input parameters.

Indirect tables are the one exception to call by value. The address of any simple table (by use of the CORAD functional modifier) may be passed as an actual parameter to an indirect table formal input parameter, and hence is a call by value. Assignment to the formal parameter effects the actual parameter.

A redefinition of formal parameters would result in a major impact on the existing implementation.

---

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time. ....F

Above properties optional .....F
Above properties are fixed at run time .....F

CMS-2 does not permit the writing of generic procedures.

Implementation of this requirement would require major modifications to the existing parameter mechanism.

---

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time. ....F

Variable number of arguments possible .....F
All but a constant number of arguments have the same type .....F
Number of arguments in each call is fixed at compile time .....F

CMS-2 does not permit a variable number of arguments.

Implementation of this requirement would require major modifications to the existing parameter mechanism.

D1. The user will have the ability to associate constant values of any type with identifiers. ....T

By use of the EQUALS and MEANS substitution features an identifier can be associated with any constant.

---

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output). ....P

Literals for all built-in types .....T
Consistent interpretation in program and data .....U

Literals are provided for all atomic data types. The documentation is not clear as to the consistency of numeric constants in both programs and input or output data.

---

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values. ....T

Initial value can be specified as part of the declaration .....T
Initialization occurs at allocation scope entry .....T
No default initial values .....T

The initial value for any numeric, character, Boolean or enumeration variable or field (component of a record) can be specified as part of its declaration or can be specified separately with a data declaration statement. If not specified there is no default value. All data units are allocated at the time of their declaration. Initialization of arrays and records must be accomplished by declaring initial values for their components.

---

D4. The source language will require its users to specify .

individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types. ....P

Numeric variable range specification mandatory .....P
Fixed point variable step size specification mandatory .....T
Range and step size specifications do not define a new type .....T

For fixed point numeric variables and fields (components of a record) the maximum absolute value and the step size must be specified by declaring the maximum number of bits required to represent the numeric and the position of the binary point. The minimum absolute value cannot be specified and is assumed to be zero. The range for floating point numerics cannot be specified.

The language has a DEBUG option which permits the user to specify a complete range for each data unit. See the comments for requirement A4.

---

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a contiguous subsequence of any enumeration type. ....P

Ranges of an enumeration type are allowed .....F
No arbitrary restrictions on the structure of data .....P

The values which can be associated with a variable or a record component (field) can be any atomic data type. A component of a record cannot be an array or another record. A component of an array can only be a record or an atomic data type. (See the comments on requirement A2.)

---

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables. ....P-

Recursive and network structures provided .....F
Handles variable-value and structure-component connections .....F
Pointer property is an attribute of a typed variable .....F
Pointer property not for constants, affects only assignment .....F
Pointer property mandatory for dynamic allocation .....F
Allocation scope never wider than access scope .....F
(Either the value or the pointer is modifiable) .....F
(Pointer mechanism handles procedures and parameters) .....F
(Remap and replace assignment have different syntaxes) .....F
(Built-in dynamic variable creation) .....F
(Variable equivalence classes are declarable) .....F

There is a very limited pointer mechanism available by use of the intrinsic CORAD function. The address of any data unit can be referenced and assigned to any numeric variable or field (component of a record). However there is no mechanism for indicating that the value of a variable is an address. The CORAD function is used primarily for indirect tables.

The full impact of the implementation of a pointer mechanism cannot be ascertained without a full description of the syntax and semantics. However, any implementation of a pointer mechanism would be a major modification to the compiler.

E1. The user of the language will be able to define new data types and operations within programs. ....P

New operations are definable only by the use of functions. Since almost all languages have functions it is assumed that the use of functions as new operations definitions is not the intent of TINMAN.

New types are definable by only enumeration (status variables) or by records (items of a table). See the comments on requirement A2.

Adding the capability to define new operations to the language beyond the use of functions would recuire major modifications to the language and its implementation.

---

E2. The "use" of defined types will be indistinguishable from built-in types. ....F

The only definable types, enumeration and records, are also built-in. Although, as stated, the requirement could be considered to be literally met, the intent of Tinman is not.

---

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations. ....P

Local variables need not be declared if OPTIONS MODEVRBL has been declared, and the type of a variable (or component of a record) may be omitted. See the comments on requirement A1.

System and local indexes, when declared, are defaulted to signed integer types with a length of 16 bits. Removing system or local indexes would be a trivial modification. Allowing system and local indexes to be declared any type would be a modest change to the language and its implementation.

---

E4. The user will be able, within the source language, to extend existing operators to new data types. ....P-

The only defined types permissible are the enumeration type and the record. Assignment can be extended to both of these types. In addition, all of the comparison operations can be applied to enumeration types.

---

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented. ....F

Construction .....F
Selection .....F
Predicates .....F
Type conversions .....F
Operations and data can be defined together .....F

There are no definable operations in CMS-2.

---

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time. ....P

Enumeration .....T
Cartesian products (records) .....T
Discriminated union .....F
Powerset of an enumeration type .....F

The only ways to define new types is by enumeration (status variables) and by records (items of a table). Implementation of discriminated union and powerset of an enumeration type would require a major modification to the language and the compiler.

---

E7. Type definitions by free union (i.e., union of

non-disjoint types) and subsetting are not desired. ....P-

Free union is permitted using the overlay features. Subsetting is only permissible on arrays (sub-tables).

The overlay capability can be easily removed.

---

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type. ....F

Initialization .....F
Finalization .....F
Allocation actions .....F
Deallocation actions .....F

CMS-2 does not provide any encapsulated means of defining initialization, finalization, allocation or deallocation actions for defined types.

Any other means of defining actions would require a major modification to the language and the compiler.

F1. The language will allow the user to distinguish between scope of allocation and scope of access. ....P-

The scope of allocation, or lifetime, of all data structures (except local indexes) is the entire life of the program. The scope of access for data structures defined in system-data-designs is the entire program, i.e., the data structures are common to all procedures. The scope of access for data structures defined within local-data-designs is the system-procedure which contains the local-data-design, i.e., the data structures are common to all procedures within the system-procedure.

A local index is defined at the beginning of a procedure and exists throughout the procedure. Its scope of allocation and scope of access are identical.

What is missing in CMS-2 is the capability of limiting the scope of access for any data structure to a procedure or to any lower level. The changes to the compiler necessary to incorporate this capability would be a moderate modification to the symbol table and the symbol table accessing mechanisms.

---

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components. ....P-

Allowable operations can be limited .....F
Access can be limited where used .....P-
External declarations need not all have the same scope .....F
Naming conflicts can be avoided (renaming) .....F

Access to data structures or program structures is limited by the location of the data or program structure definition. Data structures are either global to the entire program, local to the system procedure or local to a procedure (local indexes). See comments under requirement F1. Access rules cannot be specified either where the structure is defined or where it is used, except that it is possible to define a structure (either procedure or data) as being global to the entire program, by use of the EXTDEF modifier, even though its location would make it local to the system-procedure.

The modifications necessary to define access where the structure is called cannot be assessed at this time.

---

F3. The scope of identifiers will be wholly determined at compile time. ....P+

The scope of identifiers is wholly determined at compile time. An identifier that is global cannot be duplicated at a local level. However, an identifier that is local to one system-procedure and an identifier that is local to another system-procedure can have the same name.

Lexically embedded access scopes would only be meaningful if the language provided for data definitions at a level lower than system procedures. At present the only data definitions permissible within a procedure is a local-index. See the comments under F1.

---

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language. ....T

Applications oriented data, procedures, and functions can be stored in common libraries that are easily accessible in the language.

---

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages. ....P

Program component libraries accessible at compile time .....T
Libraries can contain foreign language routines .....U
Interface requirements checkable at compile time .....F

Program components and data in either source or object form or compools can be selected from compile time accessible libraries. Provided that the object output from foreign language compilers is compatible with the CMS-2 UYK-7 loader there does not appear to be any reason why a library cannot contain a program component that had been originally written in a foreign language.

---

F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program. ....P+

Libraries and compools will be indistinguishable .....P+
Immediately accessible sublibraries at any level .....T

CMS-2 libraries are collections of program elements (system-procedures and system-data-designs). System-procedures may be in object or source format. System-data-designs may be in object, source, or partially compiled (compool) format. The language permits the selection of any program element that is in source or compool format. (The librarian and loader may select any element in object format.) However, if the element is in compool format the user must specify that it is a compool that is being selected.

Changing the compiler so that the user would not have to specify the kind of program element would be a minor modification. From the user's point of view the compool would be indistinguishable from any other program element.

---

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware. ....P-

CMS-2 provides a limited high level I/O capability and interface with hardware. See the comments on requirement B10.

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling. ....P

Sequential execution .....T
Conditional execution .....T
Iteration .....T
Recursion .....F
(Pseudo) parallel processing .....F
Exception handling .....F
Asynchronous interrupt handling .....F
Control structures from a small set of simple primitives .....P-

Sequential execution: Unless a statement involves a branch the next statement to be executed is the statement immediately following. In addition, statements may be connected with a THEN, as in: IF <condition> THEN <statement> THEN <statement> ... ELSE... or <statement> THEN <statement> ... $.

Conditional execution is provided for by the IF <condition> THEN ... ELSE structure, the case structure (FOR <expression>), and switches.

Iteration is provided for by the VARY block structure.

Exception handling: abnormal exit parameters and overflow condition.

No syntax extension capabilities exist in CMS-2 to build new control structures. More powerful, and/or special purpose control structures have been provided, traditionally, by the addition of new primitives to the language, e.g., FIND is a table searching control structure combining iteration and a conditional. A high level macro capability could be implemented for a moderate effort.

---

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition. ....P

The language provides a GOTO operation. However, in CMS-2Y the GOTO label can be to any label within the system-procedure. Later versions of CMS-2 (such as CMS-2M) restrict the GOTO to procedure scope. In addition, the language permits switches.

Imposing that GOTO labels be restricted to procedure scope (or to any other scope rules) and eliminating switches would require only a minor modification to the language.

---

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives. ....P+

Based on Boolean expression .....T
Based on type from discriminated union .....F
Based on computed choice among labeled alternatives .....T
All alternative must be accounted for .....F
Simple mechanisms will be supplied for common cases .....P

Conditional control is provided for by the IF-THEN-ELSE, the case statement (FOR) and switches. The IF-THEN-ELSE is a common case for selection of the alternatives based on the value of a Boolean expression. The case statement and switches provide for a computed choice among labelled alternatives. For all of the conditional control mechanisms the language does not require that all alternatives be accounted for (e.g., an ELSE is not required to be specified after the IF THEN). Discriminated union is not permitted by the language.

The case statement was added to the language to provide the same capabilities as the switch mechanism in a more structured form and to localize the scope to the structure. The switch mechanism can be removed with relatively little impact.

---

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted). ....P+

Termination can occur anywhere in the loop .....P+
Multiple terminating predicates are possible .....T
Entry permitted only at the loop head .....F
Simple cases are clear and efficient .....T

Control variable is local to the loop .....P-
Control value is efficiently available after termination .....P

Iterative control is provided for by the VARY statement. Termination can be aftter a fixed number of iterations, at the beginning of an iteration upon failure of a test (WHILE), or at the end of an iteration upon failure of a test (UNTIL). The only termination of the loop permitted within the loop is by use of a GOTO outside the loop. Termination of any iteration within the loop and commencement of the next iteration can be accomplished by a RESUME statement. In addition, the RESUME statement can be used outside of the loop if the loop was terminated by a GOTO outside the loop.

Entry is permitted into the loop at any labelled statement.

If a data unit is specified as the loop control variable the data unit is global to the loop structure and hence has a value after loop termination that is available. If no data unit is specified as the loop control variable then the number of iterations is not available at loop termination as no loop control variable is used.

If the loop is terminated by a GOTO the value of the loop control variable (if there is one) is available unless it is explicitly modified. Modification of the language to make loop control variables and statement names within the loop local to the loop (so that entry is permitted only at loop head) would require a moderate modification to the scope rules and their implementation mechanisms. See the comments under F1.

---

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure. ....F

No recursive procedures within recursive procedures .....N/A
(Maximum depth of recursion can be specified) .....N/A
(Recursiveness must be specified) .....N/A

The language does not permit recursion.

Procedure linking in present implementations of the language is accomplished without the use of a procedure linking mechanism. The return address for the calling procedure is saved in the called procedure. Since there is no data that is local to a procedure (except for local indexes) there is no data allocation or management that must be performed with each procedure call. These features permit CMS-2

programs to execute without requiring the overhead of a monitor or operating system. If the language were modified to permit procedures to be recursive, an external procedure linking mechanism with a stack for the returns and for data that must be preserved (such as local indexes and loop control variables) would be required. Without changing the scope rules to comply with requirements of Section F, the external procedure linking mechanisms to provide recursion would be a minor addition to the present implementation. However, recursion without variables local to the procedure may not be of much value. If the scope rules are changed a more extensive redesign effort would be required.

---

G6. The source language will provide a parallel processing capablity. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution. ....F

Able to create and terminate parallel processes .....F
Process can gain exclusive use of resources .....F
No parallel routines within recursive routines .....F
No routines within parallel routines .....F
Maximum number of simultaneous instances are declarable .....F
(Access rules are enforced) .....F

CMS-2 provides no capabilities for parallel processing.

Extensive modifications to the language would be required to include parallel processing capabilities.

---

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program. ....P-

Program can get control for any exception .....F
Parameters can be passed .....P
Can get out of any level of a nest of control .....F
Can handle the exception at any level of control .....F

The only exception handling capabilities provided for in the language are the abnormal exit parameter (see C7), the overflow detection, the parity check, and the range check on an item index into a table.

The evaluation of an arithmetic expression in an assign operation can be tested for overflow and control transferred to a label. (SET <data unit> TO <expression> OVERFLOW <label> $).

The Boolean condition in a conditional control statement can be a test for odd or even parity on any data unit (contained in one word). (IF <data unit> EVENP (or ODDP) THEN ...)

The Boolean condition in a conditional control statement can be a test that an item index into a table is within range. (IF <table index> VALID (or INVALID) THEN ...).

The extensions to the compiler necessary to provide a general purpose exception handling control structure would be moderate.

---

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation. ....F

Priority specification .....F
Synchronization via wait/enable operations .....F
Wait for end of real time interval .....F
Wait for end of simulated time interval .....F
Wait for hardware interrupt .....F
(Can enable and disable interrupts) .....F

None of these control features are provided for in the language.

A moderate extension to the language would be required to incorporate these features.

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous. ....P

Free format with statement terminator .....T
Mnemonic identifiers possible .....T
Based on conventional forms .....P
Simple grammar .....P
No special case notations .....P
No abbreviations of identifiers or keywords .....P+
Unambiguous grammar .....P

The language is free format. All statements must terminate with a '$'. Identifiers can be from one to eight alphanumeric characters.

With some exceptions the language is based on conventional forms: Assignment is SET <data unit> TO <expression> $, the case statement uses the keyword FOR.

The grammer is simple for the commonly used features. However, there are many special purpose and seldom used features each with its own syntax. Complicated expressions involving Boolean operators and relational expressions can be unclear. There are some features that are context dependent. For example, arithmetic expressions in EQUALS substitution declarations are evaluated using a different set of precedence rules than those used for the evaluation of other arithmetic expressions.

The language could be changed easily to remove the seldom used features and special uses and to resolve the ambiguities. The implementation of these changes would be a modest effort.

---

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences. ....T

The user cannot modify the syntax of the language. However, injudicious use of the MEANS text substitution mechanism may have the appearance of syntax modification. For example, if an entire statement is substituted, the substitution name would have the appearance of a procedure call. Since a substitution name is not easily distinguishable

from a procedure name the use of MEANS text substitution can add to the visual ambiguity of the language.

The MEANS text substitution mechanism can be easily modified so that the substitution name can be easily distinguished from any other name by including a special signifying character in substitution names.

---

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset. ....T

CMS-2 uses the 64 character ASCII subset.

---

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals. ....T

Break character exists .....F
(Literals are self-identifying as to type) .....P
(Bit-string literals for any type) .....F

(1) Identifiers are formed by one to eight alphanumeric characters. The first character must be alphabetic. No break character is permitted.

(2) a. Character string literals are formed by enclosing the string in parentheses and preceding the literal by an 'H', e.g., H(ABC). Separate quoting of each line of a long literal is not required: H(THIS IS A LONG LITERAL STRING). Any break character could be used since the literal is enclosed. It would be interpreted as part of the character string literal.

b. Numeric literals may be octal or decimal. In the default compiler mode octal numbers must be enclosed in parentheses and preceded by the letter O. All numbers not enclosed in parentheses are assumed to be decimal. The compiler mode may be declared to be octal, in which case all decimal numbers must be enclosed in parentheses and preceded by a D (or may be unenclosed and suffixed by a D). All other numbers are assumed to be octal. All numbers may contain an optional radix point and may be expressed in scientific ('E') notation. No break character is permitted.

A break character of space, underline, or tilde could be used with any of the numeric literals. Only a minor modification to existing implementations would be required.

A break character of space could not be used with identifiers without introducing syntactic ambiguities into the language. Any break character to be used with identifiers would not be of much value unless identifiers could be formed from some number of characters significantly larger than eight. Adding this feature to the language would require a restructuring of the symbol table in existing compilers, thereby reducing the number of symbols available to the user.

---

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings. ....F

The language permits lexemes to continue across lines.

This capability could be removed with a relatively minor modification to existing implementations. However, a break character must be provided for long character string literals.

---

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used. ....P

The language has 179 primitives (or key words). A primitive cannot be assigned as an identifier (i.e., the primitive is reserved) if ambiguities would be created in the user's program. There are 29 primitives that cannot create ambiguities and, therefore, are not reserved.

Making all primitives reserved would require a simple modification to the existing implementation.

The large number of primitives in the language is indicative of the variety of syntax available. The number of primitives could be reduced by removing from the language those control mechanisms that are redundant or seldom used.

---

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs. ....P

Uniform comment convention .....F
Look different from code .....U
Bracketed by one or two characters .....P
Can contain any characters .....T
Can appear anywhere reasonable .....T
Terminated by the end of the line .....F
Compatible with automatic reformatting .....T

The language has three comment conventions:

(1) Comment statement: COMMENT <string> $ The comment statement cannot appear within any other statement.

(2) Bracket notes: Used to document one of the five bracket declarations (system declaration, system data declaration, system procedure declaration, head declaration, and local data declaration) <declaration><string> $.

(3) Notes: strings delimited by pairs of consecutive single primes. They may appear in any statement wherever it is permissible to use a space. ''<string>''.

Since the notes convention could accomplish the functions of the other two comment conventions it is not necessary to have more than one. The other two conventions could be removed easily from the language with no impact on existing implementations.

---

H8. The language will not permit unmatched parentheses of any kind. ....T

The language does not permit unmatched parentheses.

---

H9. There will be a uniform referent notation. ....P+

Almost all function calls and array element references use the same notation. The only exceptions are the two intrinsic functions BIT and CHAR. It can be argued that BIT and CHAR are not functions in the normal sense that the term is used, but rather operators or modifiers.

Subsequent versions of CMS-2 have adopted the uniform function notation for BIT and CHAR. This has required only a minor modification to the language and has had a reverse impact on the implementation. That is, implementation for subsequent versions required less code since one fewer syntactic form was necessary.

---

H10. No language defined symbols appearing in the same context will have essentially different meanings. ....P

There are several areas in the language where the meaning of a symbol is context dependent:

(1) AND, OR and COMP are used both as logical connectors between Boolean expressions and as bit string operators (e.g., X AND Y results in the bit by bit logical AND of X and Y). This dual use has created problems of ambiguity

(2) THEN is used both as a statement connector (<statement> THEN <statement>) and in the conditional control structure (IF <expression> THEN ... ELSE).

(3) The symbol period is used both as a label terminator and as a radix point. In addition, two periods is used for the scaling specifier and three periods is used to indicate a series of consecutive items in an INPUT or OUTPUT statement.

(4) There are three types of EQUALS declarations: numeric constant, the difference between the allocations of two addressable names, and the allocation of an addressable name. The type of EQUALS is dependent on the evaluation of the expression in the declaration. In addition, the expression is evaluated left to right with no hierarchy of operators. Numeric expressions in other contexts are evaluated hierarchically.

Elimination of any of these multiple purpose syntactic forms would require the introduction of new syntactic forms. In each case there would be no impact on other requirements and only minor modifications to the present implementations would be necessary.

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program. ....I

There are no implementation defaults that affect program logic.

---

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary. ....P

Defaults specified for don't care cases .....I
Programmer can override the defaults .....P

The language includes defaults that affect only object representation. These include:

(1) Data representation: table packing - the user may define the internal position of each field or he may let the compiler determine the most optimal internal arrangement of the fields according to one of three user specified packing modes.

(2) Function/subroutine calls: intrinsic functions may be either open or closed dependent on the input parameters. For example, the BIT modifier is open if the bit string specified does not cross a word boundary; it is closed otherwise.

(3) Reentrant vs nonreentrant: the user may indicate that the code is to be reentrant. If not specified, the code is nonreentrant. Data for reentrant code may be specified so that multiple copies are generated.

Function/subroutine calls: to provide the additional capability for the user to be able to specify whether the function is open or closed would require major code additions to existing implementations.

---

I3. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the

object machine configuration. ....P-

The only compile time capability available to the user is the CSWITCH. CSWITCH brackets allow sections of programs to be selectively compiled. The CSWITCH is set outside the program but is not available for interrogation within the program by the user. Other compile time information such as the presence of the operating system, are available only to the compiler and not to the user.

Inclusion of other compile time variables would require a major extension to the compiler.

---

I4. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled. ....P-

Conditional compilation is provided for only with the CSWITCH. See the comments under requirement I3.

---

I5. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language. ....F

The base of the language is the entire language and it is not small. There are no extension features for defining new capabilities in terms of the base.

The base of the language could be reduced to provide a relatively low level general purpose capability merely by removing those seldom used and special case features.

---

I6. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition. ....P+

Limits on the number of array dimensions, the length of identifiers, the number of nested parentheses levels and the number of nested block levels are part of the language definition. The number of identifiers in a program however, is not part of the language definition and is not fixed by the translator but is limited by the amount of core storage available on the AN/UYK-7 system that hosts the compiler.

Changing the limit on the number of identifiers in a program to a language defined limit would require a simple modification to the existing translator.

The limits on array dimensions, the number of nested parentheses levels and the number of nested block levels are set so high that no program should encounter the limits. The length of identifiers, however, could be considered by many to be arbitrarily short. See the comments on requirement H4 for the scope and impact of modifications of this limit.

---

I7. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator. ....T

Restrictions such as amount of run time storage, access to specialized peripheral equipment, etc. are not part of the language definition.

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs. ....U

No efficiency cost for unused features .....U
Efficient code can be produced for all features .....U

The current implementation of the language attempts to generate efficient code for the widest variety of programs: Sophisticated optimization techniques are used; for common special uses the compiler generates special sequences of code rather than standard sequences for all cases; only the run time support routines required by the program become part of the program. However, certain trade-offs exist within the compiler and it is not possible to ascertain that the most efficient code is generated for all programs. For example, certain features always use a run time support routine. There are probably some infrequently occurring special cases of these features for which generation of in-line code would be more efficient.

---

J2. Any optimizations performed by the translator will not change the effect of the program. ....T

Optimization by the compiler does not change the effect.

---

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions. ....T

Machine language insertions are permitted in CMS-2 programs in data designs and procedures provided the insertions are bracketed with the statements DIRECT $ and CMS-2 $. The machine language is equivalent to AN/UYK-7 assembly language.

---

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be

able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator. ....P+

Encapsulated specification of representation possible .....P
Space/time tradeoff can be specified .....P+

The language provides the user with the capability of specifying the object representation of the items of tables (records of arrays). The user can either pack the fields himself by defining the size and arrangement of the fields in the table or he can indicate how the compiler is to pack the fields by specifying a packing descriptor (NONE, MEDIUM or DENSE) with the table declaration. The packing descriptor NONE signifies that the compiler is to allocate the maximum space (and, therefore, least execution is required). DENSE signifies the least space (and, hence, most execution). In addition to the packing descriptor the user can further control the arrangement of fields within user packed tables by using the field overlay capability (free union).

With either user packing or compiler packing the user does not have complete control over the arrangement of fields within the table. A field may not start in the middle of one word and end in the middle of another word, and multi-word fields must end on a word boundary.

For user packed tables the starting word and starting bit of each field is not encapsulated but is additional information appended to the field description.

All of the fields within a table must be either user packed or compiler packed. User packing and compiler packing cannot be mixed within a table.

Expansion of the user's control over object representation of items of tables by allowing mixed user packing and compiler packing, or adding new packing descriptors, etc. would require, in addition to syntax revisions, a modification of the packing algorithms. Giving the user complete control over the positioning of fields over word boundaries would require new coding sequences and/or additional run time routines.

---

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics. ....F

Open/closed properties can be specified .....F
Open and closed versions have the same semantics .....F

All user defined procedures and functions are closed.

The current implementation of the translator could be retrofitted to include the capability of open user defined procedures/functions without requiring a major redesign. However, additional storage in the symbol table would be required to save the open procedure/function for later expansion, thereby reducing the number of symbols available to the user.

## Extraneous Features

We recommend that the following features of CMS-2, not required by the Tinman, be kept:

* Character string type (assuming that the Tinman requires only a character type, not a character string type) and the concatenation operator, CAT.

* Result parameters (OUTPUT parameters) for procedures.

* Control of the internal structure of tables (arrays) through (1) specifying the internal arrangement of the records of a simple table (one-dimensional array) as either horizontal or vertical and (2) specifying the compiler packing algorithm to be used.

* The ability to declare system procedures (modules) to be reentrant.

The following features provide a useful capability to the CMS-2 user community. They should be either kept intact or replaced by some equivalent feature:

* Bit string type. A datum of any of the atomic types can be converted to a bit string by use of the BIT functional modifier.

* The debug option. This capability has proved useful and could even be expanded.

* The MEANS statement (temporary source substitution). This could be included in a more general macro capability.

* The explicit conversion operators BIT and CHAR, which convert data to bit string type and character string type respectively, regardless of their declared type.

* The scaling specifier for intermediate computations in fixed-point expressions.

* Multiple receptacle assignment statements. The most common use for this feature is initialization at what would be the beginning of a scope in a more modern language.

* Procedure switches (item and index). Although the same effect can be obtained through a combination of a FOR (case) statement and procedure calls, this syntax enables the compiler to generate code which is

efficient in both space and time, particularly if parameter passage is involved. NTDS takes advantage of this.

The following features are highly machine dependent, but they have been found useful by CMS-2 users. They should be retained because of this user interest, but they should be required to be bracketed (encapsulated) by statements which mark them as being machine dependent (requirement J3):

* Designation that a procedure parameter is to be passed in a specific machine register.

* Designation that a datum is to reside permanently in a machine register within its scope of definition (system and local indexes). This might even be expanded to allow various data types to reside in these registers; currently only one range of integer type is permitted.

* The shift operators, which duplicate the machine shift instructions.

* The stop operators, which duplicate the machine STOP instructions. A variant of this capability also occurs in the RETURN statement.

* The COUNT function, which counts the number of 1-bits in a data unit.

The following features of CMS-2, not required by the Tinman, should be deleted:

* The EXCHANGE statement (permanent source substitution). This is actually a function of a source file editor and should not be in the language.

* OVERLAY, a form of free union.

* Sub-tables (declared subdivisions of tables). This has most of the bad properties of free union with very few offsetting virtues.

* The FIND statement (table search). This is simply a replacement for a combination of a VARY and an IF statement.

* Statement switches (index and item). These are elaborations of the GOTO statements.

* The ability to specify that a floating point datum is to be rounded. This is too hardware dependent.

* The PACK statement. This is little used and is almost useless because its inverse (UNPACK) is not in the language.

* The SWAP statement. This is too little used to be worthwhile.

* Table word referencing, which permits addressing a machine word within a table (array). This is too hardware dependent, thereby seriously inhibiting portability, and is a means of defeating type-checking as well.

## Summary

CMS-2 is an old language. The original design of the first version of the language was influenced by its predecessor (CS-1), the language concepts prevalent in the mid-sixties, and its first target machine (the AN/USQ-20). Changes to the language through the years were made primarily by the addition of new features rather than by revision. The design of the version of CMS-2 being evaluated here, CMS-2Y(7), was influenced by its predecessors, retaining most of their features and syntax, and by its target machine, the AN/UYK-7. The growth of CMS-2Y(7) has been primarily by the addition of new features. The most recent growth was caused by the current flurry over structured programming.

CMS-2 now contains most of the structured programming features common to modern HOLs: The case statement, WHILE and UNTIL conditions on iteration, block structuring, etc., in addition to those features that were considered necessary for embedded military systems: Machine code insertions, separate element compilation, compools, procedure linkage without a required monitor, mapping of structures to the machine, high level access to machine functions and hardware, and features chosen for their high efficiency (both in execution speed and space), such as the procedure switch. Some of the features make CMS-2 highly machine dependent and are contradictory to the philosophy of DOD-1.

As one would expect from a language that is an amalgam of features advocated by different schools of language design, there are some inconsistencies and context dependencies, although through careful design these have been kept to a small number. Similarly, an attempt has been made to avoid unnecessary complexity of the syntax and semantics. Although large, the language is not unwieldy or difficult to learn. In subsequent versions of the language some of the seldom used features have been culled and some of the syntax has been redefined to resolve inconsistencies and create a more uniform notation.

CMS-2 meets, at least partially, many of the Tinman requirements, primarily those that would be met by most high order languages in current use for military systems. It has a wide variety of data types and structures, including records, arrays, and fixed point, floating point, character string, and enumeration types. It has literals for most data types (not for Boolean, however!) and allows data initialization. It has all the standard arithmetic, relational, and logical operators. It has the standard control structures: Sequential, conditional, iteration, and some exception handling. It is free form, allows mnemonic identifiers and labels, has adequate commenting capabilities, and has access to libraries. It has functions with multiple inputs and procedures with multiple input and output (result) parameters.

What is missing from the language are many of the more modern features which allow a higher degree of programming sophistication. These include extension capabilities, recursion, parallel processing,

pointers, generic procedures, and suitable I/O capabilities and scope rules. The I/O capabilities in CMS-2 are very rudimentary, designed primarily for batch operation. All I/O for embedded systems is performed using machine language inserts which interface with special purpose executives tailored to the specific application. CMS-2 does not permit the scope of data units to be local to a procedure, a block, or a control structure.

It is not clear that using CMS-2 as a starting point for creation of a DOD-1 language would be the most expedient approach. It seems likely, pending more precise definitions of the features to be added, that CMS-2 could be modified to more closely meet the Tinman requirements, but the modifications would doubtlessly be extensive and costly. The resultant language probably would not be as simple and concise as desirable. Certainly it would no longer be CMS-2.

# A COMPARISON OF

# CORAL 66

# to

# TINMAN

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

## Introduction

This report gives a comparison of the language CORAL 66 to the Tinman language requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1976, Section IV). For the purposes of this comparison, CORAL 66 is considered to be defined by:

Inter-Establishment Committee on Computer Applications
Official Definition of CORAL 66
Ministry of Defence
Her Majesty's Stationary Office
London, 1970

Tinman contains 78 language requirements. This report compares CORAL 66 to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used. (Occasionally no text is needed if a requirement is totally satisfied.) If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - Only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

(The symbols P, P+, and P- will often be used with requirements which are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

The report concludes with two summaries. The first is of the features of CORAL 66 which are extraneous to Tinman and the desirability of retaining each of them. The second is of the language as a whole and the desirability of modifying it to bring in into line with the Tinman requirements.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs. ....T

CORAL 66 requires an explicit specification of the type of each variable and component of data tables. Although it allows implicit conversions during expression evaluations, the type or mode of evaluation is determined at compile time and does not change at run time.

---

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators. ....P+

Integer .....T
Floating Point .....P+
Fixed Point .....T
Boolean .....F
Character String .....P
Arrays .....T
Records .....T

CORAL 66 provides for integer, floating, fixed, and array type of data but does not provide for Boolean variables. The official definition of the language does not allow character data declarations, but Blandford extensions to CORAL 66 permit BYTE declarations which are used for character definitions and string manipulations. The TABLE declaration allows the definition of the type of a table's components including their width and length. This is equivalent to defining a record. The implementation of floating point is optional and is dependent upon the hardware. If the computer does not have the floating point unit, this feature is not required.

Introduction of Boolean type variables will require changes to existing implementations in type checking, table declarations, comparison operations, function and procedure parameter definitions etc. The floating point type data definitions can be made mandatory since this feature can be provided by software at the cost of some storage space. This may be significant on mini computers, but will cause

minimal problems on larger computers. The Blandford extension can be merged with the language kernel and made part of the official definition, instead of being left as an extension.

---

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code. ....P

Global arithmetic precision specification mandatory .....F
Individual variable precision specification permitted .....T

CORAL 66 does not allow global specification of the precision for floating point arithmetic. In fact it permits no precision specification at all for floating point numbers. They are confined to one full word.

The language does permit individual precision specification for fixed point numbers and thus meets the second part of this requirement.

To meet this requirement the language will have to be modified in following ways: (1) Implementation of floating point data will have to be made mandatory, (2) the syntax of the language and the data definition will have to be modified to permit specification of precision by the user, and (3) global specification of precision for floating computations will have to be allowed.

These changes to the language will impact all existing implementations since none of them permits global specification of precision for floating numbers. Besides, many existing implementations do not have the floating point option at all.

---

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler. ....P+

Treated as exact quantities .....T
Range and step size determined at compile time .....P
Scaling handled automatically .....P

The requirement implies that the user should be able to specify explicitly the range and step size of the fixed point numbers. CORAL 66 allows the user to specify the total bits and the fraction bits associated with a fixed point number, thus implicitly setting the possible range of values and the step size, but does not allow an explicit declaration to do so. Furthermore, it dies not allow the number to exceed one full word of the computer, thus making the fixed point data definition and its range machine dependent. Scaling for each variable is specified by the user, but is automatically handled by the compiler while processing expressions. Thus the language only partially meets this overall requirement.

The language definition should be modified to allow the user to specify the range and step size of fixed point variables. Furthermore, the language should not restrict the size of a fixed point variable to one computer word. The user should control the size of the word and its range and not the computer word size or the compiler.

Such modifications to the language will impact all existing implementations significantly since they all provide for one computer word to define a fixed point and floating point number.

---

A5. Character sets will be treated as any other enumeration type. ....F

New sets can be defined as enumeration types .....F
ASCII and EBCDIC are provided .....F
(Conversion capability between sets is available) .....F

CORAL 66 does not permit definition of a user-specified character set. The character set is defined to be ISO (Same as ASCII). Furthermore, no facilities are provided in the language to alter the collating sequence of these characters at compile time. The conversion capabilities between EBCDIC and ASCII are dictated by implementations and not required by the language definition. Thus CORAL 66 fails to meet this requirement.

Introduction of a capability to enumerate a character set in CORAL 66 is a major language design change having an equally significant effect upon implementation. It would require adding status variables and enumeration capabilities which the language does not presently have.

Changes in the language to accommodate user defined character sets will require modification to all existing implementations. The library will have to include routines to convert ASCII to EBCDIC and other installation and implementation related conversion routines.

Other features of the language related to strings and character sets will also be affected to accommodate the programmer-defined character set.

---

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope. ....P+

Number of dimensions is fixed at compile time .....T
Type is fixed at compile time .....T
Lower subscript bound is fixed at compile time .....T
Upper subscript bound is fixed at scope entry .....F
Subscripts only integers or from an enumeration type .....T
Subscripts will be from a contiguous range .....T

CORAL 66 allows one or two dimensional arrays each element of which is of the same type. It requires that the lower and upper bounds be supplied by the user in the form of integers, and thus be determinable at compile time. The language definition does not allow the computation for the upper bound to be deferred until scope entry time. It must be available at compile time.

To defer processing of upper subscript bound implies provision for dynamic array sizes. This would mean that storage allocation for the array would be done at the run-time. It would require more time and care to execute the programs having dynamic arrays. Modifications to the translators will have to be made to alter the existing syntax-checking mechanism (which checks for the numerical value of both upper and lower bounds), the storage allocations functions (which allocate core for arrays at compile time), and the compile time diagnostics.

---

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time. ....P

Alternative structures for records are possible .....T
Discrimination condition may be any Boolean expression .....F

The OVERLAY feature of CORAL 66 causes apparently different data references to refer simultaneously to the same objects of data (i.e., as alternative names for the same storage locations.) This feature can be used to overlay, and later access, variables and array or record components. However, there is no explicit discrimination condition available to the user for accessing the record.

To meet this requirement the language will have to provide a mechanism to allow the user to evaluate a Boolean expression and then use one or the other form of record in his program.

Introduction of this feature will require changes to all existing implementations to allow the OVERLAY feature to work in conjunction with the discrimination condition.

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value. ....T

Automatically defined for any type (except...) .....T
Available for individual components .....T
(Assignment and reference via functions) .....F

Assignment to integer, floating and fixed data types is available in CORAL 66, as is the user ability to reference the values stored. Components of arrays and records can also be individually assigned to or referenced. However, the assignment operation is not defined for entire records or arrays, even if they are conformable.

No conflicts with other requirements.

---

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity. ....T

CORAL 66 contains = operator which can be used to compare two logically equivalent values of different types for equivalence. Thus in a logical comparison floating 1.0 will be equal to integer 1.

No conflicts with other requirements.

---

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration. ....P-

Built-in for all numeric and enumeration types .....P
Ordering can be inhibited when desired .....F

CORAL 66 provides for relational operators <, <=, :=, >=, <> and > for logical comparison of numeric data. However, since CORAL 66 does not permit status variables, automatic ordering of enumerated data is not permitted. Unordered definitions of sets is also not allowed.

To meet this requirement, CORAL 66 will have to define STATUS variables with the capability to define their associated set of values either in an ordered or unordered way. Introduction of this type of variable will affect the existing implementations significantly. All logical expression processing will have to be modified. Its impact on control statements will require changes in IF and FOR-WHILE statements also.

All existing implementations will have to be modified to include the Status variables and changes to the constructs mentioned in the preceding paragraph.

---

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation. ....P+

Addition .....T
Subtraction .....T
Multiplication .....T
Division with real result .....T
Exponentiation .....F
Integer and fixed point division with remainder .....F
Negation .....T

CORAL 66 does not provide for exponentiation nor does it provide for integer division with remainder. Integer division of the type K := I/J; results in the integer quotient being stored in K and the remainder is lost to the user. However if K were defined to be floating, then I and J will be first converted to floating before division takes place. The function INTEGER has to be used in such cases to force division before conversion (i.e., K := INTEGER(I/J);). Operators for addition, subtraction, and negation, with appropriate conversions where necessary, are also provided.

Implementation of exponentiation can be done via software subroutines. Many CORAL 66 compilers already have this feature. Integer division with remainder will require some syntax changes in the language to enable the user to specify where the remainder has to be stored. Both of these features can be implemented with minimal impact on existing translators. Areas impacted will be syntax analysis, expressions, diagnostics and library.

---

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers. ....F

Never from the left for data within range .....F
Never on the right for integer and fixed point .....F
Implicit floating point rounding beyond precision allowed .....F
(Run time checks can be avoided) .....N/A

The CORAL 66 language does not provide explicit rules or functions for truncation from the right for integers or fixed point numbers nor does it prohibit truncation from the left. This matter is left entirely in the hands of the implementors. On the floating point numbers, the user has no option to specify the precision. Each floating number occupies one computer word. Their level of precision, and the precision of intermediate results is determined by the hardware or the implementors. Hence the language fails to meet this requirement.

The CORAL 66 definition can be modified to include rules for truncation and rounding. Users should be allowed to specify precision for floating point numbers. Implementation of these changes can be accomplished with minimal effort. It is conceivable that some of the existing compilers may not be allowing truncation from the left. Other features impacted as a result of these changes will be multiplication, addition, assignment, and expression processing.

The existing implementations will have to ensure that these rules for truncation and rounding are also followed to compute and store the intermediate results.

---

B6. The built-in Boolean operations will include "and", "or", "not", and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode. ....P

Short-circuit and .....T
Short-circuit or .....T
Not .....F
Xor .....F

CORAL 66 requires short circuit evaluation of the AND and OR Boolean operators. It does not provide for NOT or XOR.

The introduction of NOT and XOR Boolean operators will have minimal impact on expression evaluation and some control statements requiring logical conditions (e.g., IF, FOR-WHILE etc.). The existing implementations will have to be modified accordingly.

---

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure. ....F

Scalar operations on arrays .....F
Assignment between records and arrays of conformable type .....F

The CORAL 66 language definition does not specify that scalar operations be extended to conformable arrays or that data transfers be permitted between records or arrays of identical logical structure. Some implementations of CORAL 66 provide some of these facilities, but that is an accident of implementation, not a language specification.

This feature is not difficult to implement. It would require further type checking for arrays and components of records, their conformability for size and subscript ranges before generating code to perform the scalar operations or data transfers for the entire array or records.

---

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors. ....P-

No implicit conversions .....F
Explicit between integer, fixed point, and floating point .....T
Explicit between fixed point scale factors .....F
(Explicit between integer and Boolean) .....F
(Explicit between integer and enumerated types) .....F
(Explicit between different enumerated types) .....F

During expression processing the CORAL 66 language definition permits implicit type conversions. However, explicit conversion operators between integer, fixed, or floating point data are also available. Any expression may be enclosed in parentheses and be explicitly typed by a prefix 'INTEGER', 'FIXED' scale, or 'FLOATING'. For example

```
'FLOATING'(N + 'INTEGER'(X*Y - A/B))
```

is a primary of type floating.

To introduce and further emphasize the explicit conversions in the language, the current language definition and implementations will have to be modified to forbid implicit conversion. Some modifications to the syntax-checking procedures will also have to be made to accommodate explicit conversions between two fixed-point scale factors.

---

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated. ....P-

Implicit conversion between ranges .....F
Exception condition on integer and fixed point truncation .....P

There are two kinds of ranges: the range of variables and the range of subscripts. CORAL 66 does not permit specification of range of variables, hence the question of implicit conversions between ranges does not arise in this case. Array size ranges are required in CORAL 66, but they must be supplied by the user as integer constants and are compile time evaluable. The definition seems to imply that an error message will be issued to the user if he supplies other than an integer constant as a subscript range. No automatic conversion is implied.

The language does not require that an exception condition be raised if an integer or fixed point number is truncated.

The language does, however, permit a run-time mode of operation in which exception conditions are raised if the subscript range is exceeded.

To meet this requirement CORAL 66 compilers may have to be modified to allow subscript ranges of types other than integers, and also to permit automatic conversions or truncation of these ranges to integers. The concept of ranges for all variables will have to be introduced and the language syntax modified to accomplish this.

---

B10. The base language will provide operations allowing programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent. ....F

Sending and receiving of data .....F
Sending and receiving of control information .....F
Dynamic device assignment .....F
User exception condition control .....F
Installation independence .....F
(Data formatting capability) .....F
(Reading and writing of bit strings) .....F

The CORAL 66 language makes no provision for I/O statements as part of the official language definition.

Language features to select a device, open and close a file, read and write data, handle interrupts, etc. should be defined. An I/O package, similar to the one currently available on the GE4080 computer at the Royal Radar Establishment, should be used (with necessary modifications) to carry out the functions required by the I/O statements.

---

B11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate. ....P-

Union .....T
Intersection .....T
Difference .....T
Complement .....F
Membership predicate .....F
(Set inclusion) .....F

CORAL 66 permits the operations of DIFFER, UNION and MASK to perform the logical operations of difference, union and intersection respectively. It does not provide for other logical operations specified in Tinman requirements. These operations are only defined for the existing types since CORAL 66 does not allow user-defined data types.

To fully meet this requirement the language will have to (1) allow user definition of new data types and (2) allow all of the powerset operations specified above to be applicable to both the built-in data types as well as the user defined data types. This would require major changes to the current definition of the language as well as to the existing implementations.

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right. ....P-

Side effects must occur in left-to-right order .....P-
(Embedded assignments) .....F

CORAL 66 definition does not specify algorithms for expression evaluation and leaves their implementation dependent. Only for evaluation of library built-in functions in a simple expression does the language definition state that they will be evaluated in the order in which they appear when the expression is read from left to right, regardless of brackets. The only exception to this rule is when other functions appear as value parameters of a function and hence must be evaluated before the function is evaluated (i.e., the order of evaluation of sin(cos(exp(n))) will be exp, cos, sin.).

To establish uniformity in all implementations, the language definition should specify rules for expression evaluation. Only where the order of evaluation is important should the language impose the restriction that evaluation must proceed from left to right. In other instances the compiler implementors should be given the choice in selecting evaluation rules, as long as the effect and the results of evaluation are not changed.

Such a change is likely to affect the existing implementations which have followed their own rules for the order of evaluating expressions.

---

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized. ....P

Few precedence levels .....F
No user-defined precedence levels .....T
Operands of an operation are obvious .....T

CORAL 66 does not define precedence levels for the operators. The only established precedence rule is that all syntactically outermost terms in an expression will be evaluated to the required numeric type before the adding operators are applied. If an expression is enclosed in parentheses, its terms are not outermost and the rule no longer applies. The algorithm for the particular compiler will determine the sequence of events.

A few precedence rules for operator hierarchy can be established in the language definition. This may affect some, though not all, implementations. For instance, one set of precedence rules can be: 'MASK', 'UNION', 'DIFFER', */, +-, and the ordinary priority of operators may be changed by using parentheses within an expression.

---

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed. ....T

The language allows expressions wherever both constants and variables are allowed.

No conflicts with other requirements.

---

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time. ....F

The official definition of CORAL 66 does not permit constant expressions wherever constants are allowed. For instance, FIXED(2*8,3/1) X,Y; is an illegal data definition, whereas FIXED(16,3) X,Y; is acceptable. Similarly, the lower and upper bounds of the array in an array declaration must be a constant, and not a constant expression.

If a requirement is established in the language definition that all constant expressions must be evaluated before run time, then CORAL 66 will meet this requirement. Constant expressions, such as in the examples above, can then be legally used. This will require minimal changes to syntax checking routines of existing compilers.

---

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to

ease of learning. ....P

Parameter rules consistent in all contexts .....F
No special operations applicable only to parameters .....T

CORAL 66 allows procedure parameters, parameters for declarations, and built-in operators, but does not permit parameters for exception handling or parallel processing. While procedure parameters can be objects of data, places in the program, data references, procedure names, expressions, etc., the parameters in several data declarations (e.g., FIXED option) can only be integer constants. Hence the language is not uniform in its treatment of parameters in the sense of the Tinman requirements. However, no special operations are allowed on parameters which are also not permitted for other structures.

The language will have to provide for exception handling control (or other) features, for parallel processing and for user definition of new operators. These are all major changes to the existing language and will require significant effort to implement.

---

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter. ....P

Actual and formal parameters will agree in type .....T
Rank of parameter arrays is fixed at compile time .....T
Parameter array size and subscript range can be passed .....F

CORAL 66 requires type checking for compatibility between actual and formal parameters. It also requires that the number of dimensions, the size, and the subscript range for array parameters be known at compile time. Thus it does not allow for dynamic arrays and hence does not meet part of this requirement.

The language definition must be modified to allow for dynamic arrays and also passing of parameters related to the size and subscript ranges of these arrays. This would be a significant change for existing implementations.

---

C7. There will be only four classes of formal parameters.

For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters. ....P+

Act as constants (call by value plus) .....T
Act as variables (call by reference) .....T
Exception control .....F
Procedure parameters .....T
(Act as variables, but call by value) .....T
(Act as variables, result parameter) .....T

CORAL 66 provides for all of these types of parameters with the exception of exception handling parameters. There is no provision for exception handling in the language. However, it also provides for parameters not required by the Tinman requirements (e.g., SWITCH).

The exception handling mechanisms should be added to language features (e.g., ON OVERFLOW etc.) which allow transfer of control to labels outside the scope in which the exception condition occurs. Those types of parameters (e.g., SWITCH) which are not required by the Tinman requirements should be disallowed.

---

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time. ....P-

Above properties optional .....F
Above properties are fixed at run time .....P

CORAL 66 does not support the capability to define generic procedures in which the type of the parameters is determined at compile time by the type of the actual parameters. The type specification for each parameter on the formal side is required in CORAL 66 and is not optional as specified by the Tinman requirements. The same is true for dimensions of the arrays and scale factors of fixed point variables. CORAL 66 does not permit specification of range, precision, or format in parameters.

Implementation of this requirement will imply many changes in the existing calling mechanism of the language. Each parameter, for instance, will be followed by type specification, a range of values, precision specification where applicable, dimensions, and scale in the

calling mechanism. Some of these items are not even currently defined in the language. Additionally checking for these items will have to be implemented on both the actual as well as the formal side of the call. On the formal side the generic procedure capabilities will have to be introduced which will require selection and/or generation of code for the formal procedure for the type, mode, precision, etc. required by the actual calls.

---

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time. ....F

Variable number of arguments possible .....F
All but a constant number of arguments have the same type .....F
Number of arguments in each call is fixed at compile time .....F

This facility is not available in CORAL 66.

Implementation of this capability will affect the language and all implementations. It will require provision for variable number of parameters in procedure calls. Even the existing I/O procedures in some of the existing implementations will have to be modified to accommodate this.

D1. The user will have the ability to associate constant values of any type with identifiers. ....F

CORAL 66 does not permit an identifier to be associated with a constant, except by assignment.

Minimal changes to the language definition and existing implementations will be required to allow a variable to be declared a constant and be associated with the value of a constant of the same type.

---

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output). ....P

Literals for all built-in types .....T
Consistent interpretation in program and data .....F

The CORAL 66 definition provides for the syntax for the built-in data types e.g., fixed, floating and integer. Blandford extensions to the language also include definition and syntax for characters. But the language does not provide for I/O features nor does it specify the syntax for data to be input or output.

Several implementations (maybe all existing implementations) of CORAL 66 allow the same value for numeric constants within the program and I/O data. To meet this requirement, standardized I/O features have to be included in the language definition. Minimal changes to process those I/O statements utilizing existing I/O routines and facilities will be required in the existing implementations.

---

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values. ....P+

Initial value can be specified as part of the declaration .....T
Initialization occurs at allocation scope entry .....T
No default initial values .....F

CORAL 66 allows initialization of variables at the time of their declaration. Initialization of these variables takes place at the time of scope entry. However, there is no language specified guarantee that no default initial values will be assigned to variables not explicitly initialized by the user. Nor does the language definition specify that an error condition will be raised if a uninitialized variable is accessed before assignment of a value. The subject is implementation dependent. Hence CORAL 66 does not meet this portion of the requirement.

All existing implementations will be affected if they are to meet this requirement. It will require provisions for compile time checks to ensure that no variable is being accessed without first being given a value either by assignment, I/O, or initialization. Error diagnostics should be given if this is not the case. Furthermore, no implementation should initialize variables by default, as is commonly done.

---

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types. ....F

Numeric variable range specification mandatory .....F
Fixed point variable step size specification mandatory .....F
Range and step size specifications do not define a new type .....N/A

CORAL 66 does not allow range specifications for numeric variables nor does it permit step size specification for fixed point variables.

Implementation of this feature will require modification of CORAL 66 syntax to allow explicit range specification for all variables and step size specification for fixed point variables. The compilers will then have to generate code to check for ranges at execution time.

---

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a contiguous subsequence of any enumeration type. ....P-

Ranges of an enumeration type are allowed .....F
No arbitrary restrictions on the structure of data .....P-

CORAL 66 does not meet the first part of this requirement because it does not permit specification of any kind of ranges.

In the second part of the requirement, there is no way that records can be made part of an array nor is it conveniently possible to make arrays part of a record. However, in table declarations (which represent records) one can define an element of an array as constituting one entry in the record. Subsequent entries in the table can be made to represent subsequent elements of the array. Only in this partial sense can an array be made a component of a record.

To completely meet this requirement would require changes in the language and the implementations. The language has to be modified to accept ranges in declarations, definition of new data types, status variables, and an expanded definition of records and arrays, to permit the inclusion of arrays in record definition and vice versa.

---

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables. ....P-

Recursive and network structures provided .....F
Handles variable-value and structure-component connections .....P
Pointer property is an attribute of a typed variable .....F
Pointer property not for constants, affects only assignment .....T
Pointer property mandatory for dynamic allocation .....F
Allocation scope never wider than access scope .....T
(Either the value or the pointer is modifiable) .....T
(Pointer mechanism handles procedures and parameters) .....P
(Remap and replace assignment have different syntaxes) .....F
(Built-in dynamic variable creation) .....F
(Variable equivalence classes are declarable) .....F

CORAL 66 does not provide for a pointer data type. However, its LOCATION operator can be used to point to the location containing data values for a variable or array and to access those values. The language does not provide any safety features for this option and the programmer himself has to ensure the validity of the pointers and the data being accessed. If the pointer is pointing to the first item in a table or array, items in the subsequent words can be referenced; but if the item

is located in a part of a word, there is no guarantee that it can be referenced.

Although it is possible to structure lists using the LOCATION option, the CORAL 66 language does not permit declaration of such lists in the data definition. The LOCATION option is also inadequate to define recursive data structures.

To properly meet this requirement, the language must provide for a pointer data type capable of pointing at variables, arrays, tables, elements of tables, procedures, parameters etc., and to allow recursive data definitions and null references. It must allow for easy modification of the data value pointed to or the pointer value itself. Such changes in the language will affect all existing implementations.

E1. The user of the language will be able to define new data types and operations within programs. ....F

CORAL 66 does not provide for user definition of new data types or operations.

Special type checking mechanisms will have to be added to existing compilers to allow for recognition of new data types and operations and then their use as built-in types. The language definition will have to be modified to show how new data types and operators can be added and used.

---

E2. The "use" of defined types will be indistinguishable from built-in types. ....F

Not applicable to CORAL 66 since it does not allow defined types.

The changes needed to fulfill this requirement are discussed under requirement F1.

---

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations. ....T

CORAL 66 requires that every data item be explicitly declared before use.

No conflict with other requirements.

---

E4. The user will be able, within the source language, to extend existing operators to new data types. ....F

CORAL 66 does not allow defined data types and hence does not provide the capability of extending the existing operators to defined types.

In addition to definitional facilities specified in E1, a mechanism should also be provided to specify the operations that can be performed on the defined data types. Other language features will also be affected since their definition will have to be extended to provide for new data types and new operators.

All of these modifications specified in E1 and in the preceding paragraph will significantly impact all existing implementations.

---

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented. ....F

Construction .....F
Selection .....F
Predicates .....F
Type conversions .....F
Operations and data can be defined together .....F

CORAL 66 does not allow definition of new operations or data types, hence does not meet this requirement.

The entire capability to extend the capabilities of the language to meet the needs of different applications by allowing user to define new data types and new operations has to be added to the language. Furthermore, the user should be able to associate new operations with new data types and should also be permitted to extend existing operations to new data types. This would require changes in existing translators in the area of lexical and syntax analyzer, construction of dictionaries and intermediate language table, and the type checking mechanisms.

---

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time. ....F

Enumeration .....F

Cartesian products (records) .....F
Discriminated union .....F
Powerset of an enumeration type .....F

CORAL 66 does not support defined types at all.

The changes needed to fulfill this requirement are discussed under requirement E1.

---

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired. ....F

CORAL 66 has an overlay capability, which provides a form of free union.

To remove the overlay capability would greatly simplify existing compilers, but would have a profound effect on existing programs written in CORAL 66.

---

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type. ....F

Initialization .....F
Finalization .....F
Allocation actions .....F
Deallocation actions .....F

Since the language does not allow definition of new types, the questions of initialization, allocation, etc. of new data types does not arise.

The changes needed to fulfill this requirement are discussed under requirement E5.

F1. The language will allow the user to distinguish between scope of allocation and scope of access. ....T

CORAL 66 allows allocation of storage for variables at the beginning of procedures, in a block (e.g., between BEGIN and END statements) and via a COMMON statement. Access to the variables is controlled by language defined rules. A variable allocated by a COMMON statement can be accessed by several procedures having that variable in their COMMON declarations; a variable declared at the beginning of a procedure can be accessed from anywhere within the procedure except in embedded macros or blocks containing another declaration with the same name. A variable declared within a BEGIN and END block will only be accessible within that block. Thus CORAL 66 fully meets this requirement because it separates the scopes of allocation and access for defined data.

No conflicts with other requirements.

---

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components. ....P-

Allowable operations can be limited .....F
Access can be limited where used .....F
External declarations need not all have the same scope .....T
Naming conflicts can be avoided (renaming) .....F

CORAL 66 does not allow specification of access rules during the definition of structures (procedures, data etc.), nor does it allow specification of access rules at the place where the structure is used. At the time of structure usage language-defined access rules are followed in case of a conflict of names, but the user has no choice in the matter.

CORAL 66 also does not have a renaming facility for cases where two or more names refer to same location.

However, the current definition of CORAL 66 permits different scopes for items declared in COMMON statement. This is done at the time of variable definition.

The ability to limit access to structures both where they are defined and where they are accessed will require extensive changes in language syntax and implementations. A mechanism will have to be

specified to state the scope and access rules at the time of structure definition. Implementations will have to be modified to test for access rights each time a structure tries to access another structure.

---

F3. The scope of identifiers will be wholly determined at compile time. ....T

This is required in CORAL 66 definition. Scope rules have been specified and resolution of apparent conflicts in names is resolved before code generation time.

No conflicts with other requirements.

---

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language. ....T

The capabilities to develop application oriented libraries are available in the language. The LIBRARY option permits the use of procedures stored in the library.

No conflict with other requirements.

---

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages. ....P

Program component libraries accessible at compile time .....P
Libraries can contain foreign language routines .....T
Interface requirements checkable at compile time .....F

The libraries of CORAL 66 are not capable of holding anything definable in the language. For example, it cannot hold data definitions by themselves unless they are parts of a procedure. The use of EXTERNAL

option allows interfaces with object programs compiled from other source codes provided they are compatible with the loader. Thus the check for compatibility with these routines is made at load time and not at compile time.

Modifications to the language to define and store groups of data definitions in the library for later access by procedures is necessary to meet this requirement. This facility will have to be included in existing implementations. Furthermore, the existing implementations will have to be modified to allow compile time checks with object code compiled from other source codes instead of at the load time, as is specified in the CORAL 66 definition.

---

F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program. ....P-

Libraries and compools will be indistinguishable .....F
Immediately accessible sublibraries at any level .....?

CORAL 66 does not provide for compools. It does not allow for partitioning of libraries according to use by different projects although the effect can be achieved by the existing library facilities which allow any number of project oriented routines to be compiled and stored for later access by project programs.

The implementations will have to be modified to include the compool facilities and to make them compatible with the existing library facilities. Simple procedures must also be developed for partitioning libraries according to general or specialized use.

---

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware. ....F

CORAL 66 does not provide for standard machine independent capabilities for I/O, file management, or other peripheral usage. No attempt has been made to standardize language interface with the operating system.

This is a difficult feature to define. Standardization of language interface with the OS is still beyond the state-of-the-art. However, some language features can be defined to perform I/O functions. Changes will be required in the existing implementations to accommodate this.

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling. ....P

Sequential execution .....T
Conditional execution .....T
Iteration .....T
Recursion .....P
(Pseudo) parallel processing .....F
Exception handling .....F
Asynchronous interrupt handling .....F
Control structures from a small set of simple primitives .....T

CORAL 66 permits the normal sequential execution of statements, provides for IF-THEN-ELSE conditional control and FOR loop for iteration control. It optionally allows the RECURSIVE option to be specified for procedures. However, it does not provide for pseudo or real parallel processing, exception handling, or interrupt handling. The language has defined a small set of primitives which have been used to build these control structures.

The language needs to define control structures which allow the user to perform parallel processing, exception handling, and interrupt handling. These features will have a significant impact on all existing implementations.

---

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition. ....F

CORAL 66 allows labels to be defined in the COMMON statement and thus permits global jumps from one segment to another. Hence it does not meet this requirement.

No conflict with other requirements.

---

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a

computed choice among labeled alternatives. ....P+

Based on Boolean expression .....T
Based on type from discriminated union .....F
Based on computed choice among labeled alternatives .....T
All alternative must be accounted for .....T
Simple mechanisms will be supplied for common cases .....P

CORAL 66 allows Boolean expressions in its IF-THEN-ELSE control structure. The result of evaluation of Boolean expression in short circuit mode determines the action to be taken. The language also allows a SWITCH statement in which the control is transferred to one of the labelled variables depending upon the value of the index but not the type of the discriminated union. The language requires all alternatives in the conditional controls properly accounted for. It also permits simple mechanisms for common cases by allowing, for example, a short circuit mode evaluation of Boolean conditions in IF statement.

No conflicts with other requirements.

---

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted). ....P+

Termination can occur anywhere in the loop .....T
Multiple terminating predicates are possible .....T
Entry permitted only at the loop head .....T
Simple cases are clear and efficient .....T
Control variable is local to the loop .....F
Control value is efficiently available after termination .....T

The CORAL 66 requires that the termination condition for FOR statement in both forms (i.e., FOR with STEP or FOR with WHILE) be provided at the beginning of the loop. Each time the loop is executed, this termination condition is evaluated at the beginning of the loop. The user is allowed to use an IF-THEN or GOTO statements anywhere in the loop giving him the facility to terminate the loop. The language also allows a loop to be terminated in more than one way, permits entry only at the top and allows simple forms. The loop control variables are not local to the loop, they must be declared in the beginning of the procedure in which the loop occurs. The value of the control variable is available after the termination of the loop.

The requirement stating that the control variable be local to the loop seems to be in conflict with the requirement that all variables be explicitly declared before use. If a control variable is defined before use in a loop, it will also be defined after the termination of the loop and thus will not be local to the loop. For instance, CORAL 66 requires that all control variables be defined at the beginning of the procedure in which the loop is present. These variables are not local to the loop. On the other hand, the assumption that the construct FOR I=1 THRU 5 defines I is also wrong. FOR I=1 etc. does not really define the construct variable I. It assigns a value to the variable without defining it. The attributes of I are unknown and will have to be defined by default by the compiler.

---

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure. ....P

No recursive procedures within recursive procedures .....F
(Maximum depth of recursion can be specified) .....F
(Recursiveness must be specified) .....T

The CORAL 66 language allows both recursive and non-recursive procedures in the definition. If a procedure is not explicitly declared to be RECURSIVE, non-recursion is implied. There is no language specified restriction that a recursive procedure cannot have another recursive procedure within its definition.

Maximum depth of recursion is also not specified.

No conflict with other requirements.

---

G6. The source language will provide a parallel processing capablity. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution. ....F

Able to create and terminate parallel processes .....F
Process can gain exclusive use of resources .....F
No parallel routines within recursive routines .....F
No routines within parallel routines .....F
Maximum number of simultaneous instances are declarable .....F

(Access rules are enforced) .....F

CORAL 66 does not support parallel processing.

Extensive modifications to the language capabilities will have to be provided to meet this requirement. The language syntax will have to include facilities for task initiation, termination and resumption and will also have to provide for waiting, event related interrupt handling, etc. The implementation of these facilities in the compiler will require extensive interfaces with the scheduler, loader, interrupt handler, storage allocator, disc and file handler, and other portions of the operating system. Hardware capabilities (e.g., the available storage, single or multi-processing environment etc.) will also have to be taken into account.

---

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program. ....F

Program can get control for any exception .....F
Parameters can be passed .....F
Can get out of any level of a nest of control .....F
Can handle the exception at any level of control .....F

CORAL 66 does not include any special support for exception handling.

No conflict with other requirements.

---

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation. ....F

Priority specification .....F
Synchronization via wait/enable operations .....F
Wait for end of real time interval .....F
Wait for end of simulated time interval .....F
Wait for hardware interrupt .....F

(Can enable and disable interrupts) .....F

CORAL 66 does not support any of these capabilities.

No conflict with other requirements.

Requirement H1

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous. ....P+

Free format with statement terminator .....P+
Mnemonic identifiers possible .....T
Based on conventional forms .....T
Simple grammar .....T
No special case notations .....T
No abbreviations of identifiers or keywords .....T
Unambiguous grammar .....P

CORAL 66 very nearly meets this requirement. It allows its statements to be entered in free format anywhere in the line and terminated by a semi-colon. There are a few exceptions to free formating e.g., no blanks are permitted in composite operators like >= etc. The language does not limit the legth of the identifier names, but allows the compilers to optionally select the first 12 characters to determine the uniqueness of the identifiers. The user can thus keep the names more mnemonically significant, and the compilers can keep their syntax checking efficient. CORAL 66 syntax follows conventional forms and left to right scan and has simple grammar rules such as the parenthesis are evaluated first in any expression. It does not have special notation for rare cases, nor does it allow abbreviations of keywords in the language definition. The syntax of the language is generally unambiguous with few exceptions. For example,. example, the two constructs BIT and BITS are syntactically very close to each other and yet have significantly different usage. Similarly, the constructs AND and MASK have the same meaning in mathematical sense, but have significantly different usage in CORAL 66.

Changes to the language definition to select constructs which are significantly distant from each other to represent different concepts is necessary to meet this requirement. Implementation of such changes will be trivial although the impact on user programs may be significant.

---

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences. ....P

Although the facility to modify language syntax and operator hierarchies or the capability to introduce new precedence rules, definition of new infix operator precedence, etc. is not available or permitted to the user, the language does allow use of additional features (including keywords) not officially within the CORAL 66 language, and not clashing with the official definition or with each other, to be implemented in the compiler. Hence CORAL 66 only partially meets this requirement.

The language definition should explicitly forbid additional keywords to be introduced in the language. Extensibility should be permitted only via procedure, user data and operator definitions without modification of existing rules of the language.

---

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset. ....T

All features of the language can be constructed utilizing the ASCII character set. In certain instances the language has provided concessions to accommodate representations acceptable to ASCII (e.g., the conventional "less than or equal to" symbol is replaced by <=).

No conflicts with other requirements.

---

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals. ....T

Break character exists .....T
(Literals are self-identifying as to type) .....T
(Bit-string literals for any type) .....P

The language specifies rules for forming identifiers and literals and gives examples. It allows use of blank character in identifiers without acting as a terminator. Hence the blank can be used as a break character.

The type of CORAL 66 literals is determinable from their syntax. Octal literals are allowed, but bits are not permitted as literals.

No conflict with other requirements.

---

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings. ....T

CORAL 66 does not allow continuation of lexical units across lines.

No conflict with other requirements.

---

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used. ....T

A list of keywords is provided in an Appendix. There are only 44 keywords. They are all reserved and cannot be used as identifiers.

No conflict with other requirements.

---

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs. ....P+

Uniform comment convention .....T
Look different from code .....T
Bracketed by one or two characters .....P
Can contain any characters .....P
Can appear anywhere reasonable .....T
Terminated by the end of the line .....F
Compatible with automatic reformatting .....T

There are two ways of inserting a comment in CORAL 66: First, by starting with COMMENT and terminating with a semi-colon. In this case the text of the comment cannot contain a semi-colon. Second, by enclosing the comment within round brackets immediately following a semi-colon in a program. The text may contain brackets provided they are matched. In either form of comment, the end of line is ignored in the comment and a semi-colon terminates the comment.

To meet this requirement the COMMENT form of comment shall have to be dropped since it does not start with one or two characters. The language should allow any character, including a semi-colon, in the comment. Furthermore, the comment should be allowed to be terminated by the end of line.

Some changes will be required to all existing implementations to accommodate the language changes.

---

H8. The language will not permit unmatched parentheses of any kind. ....T

CORAL 66 does not allow unmatched parentheses in expressions or comments nor does it permit unequal numbers of BEGINs and ENDs.

No conflict with other requirements.

---

H9. There will be a uniform referent notation. ....F

CORAL 66 encloses array subscripts in square brackets and function parameters in parentheses. It, therefore, maintains a syntactic difference between function call and array element references. In addition, it does not allow functions to appear on the left hand side of an assignment statement as is permitted to arrays. Hence the language does not meet this requirement.

Implementation of identical syntax for arrays and functions is a simple fix. However, allowing functions to be treated like pseudo variables would require moderate effort to implement.

---

H10. No language defined symbols appearing in the same context will have essentially different meanings. ....T

CORAL 66 symbols are unambiguous and have the same meaning in all usages. In particular CORAL 66 differentiates between := and = and does not allow parenthesized parameters.

No conflict with other requirements.

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program. ....P

Most defaults related to program logic are specified in the manual and are under programmer control with few exceptions. For example, the SWITCH statement does not specify the out or default clause which determines the transfer of control in the event the value of the switch is either zero or exceeds the number of labels provided in the declaration. This feature is implementation dependent and can alter the flow of program control.

To meet this requirement the OUT or OTHERWISE clause should be added to the SWITCH statement, allowing the programmer to specify the default condition. Implementation of this feature is trivial.

---

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary. ....P+

Defaults specified for don't care cases .....T
Programmer can override the defaults .....P

There are some defaults that a programmer may care about and override, but there are some others which he cannot control. In the first category comes the subscript range checking which is not checked by default. But the programmer has the option to specify that code be generated to check it. In the second category is the parameter checking default for procedures. No run time parameter checking is performed and the user cannot specify that this be done.

CORAL 66 compilers have placed prime importance on run time efficiency and compact code. The language has, therefore, deliberately avoided features such as dynamic arrays and run time checking. To meet this requirement, an option should be provided to the user so that code is generated to perform run time checking of procedure parameters and other features.

---

I3. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration. ....F

CORAL 66 does not provide the facility to provide information to compilers regarding the characteristics of the object machine, the configuration, etc. so that appropriate code could be generated for various configurations.

Implementation of this capability would require selection of these "compile time variables" for the language and processing them to generate different types of code for various computers, and configuration. A determination will also have to be made as to where to place these features in the program. Implementation of these features will have minimal effect on syntax and semantic analysis phases of compilation, but will have a significant impact on the code generation phase of the compilers.

---

I4. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled. ....F

The present definition of CORAL 66 does not provide for control structures which can be used for conditional compilation.

Implementation of such control features would not impact other features of the language. The existing implementations will be affected since they will have to be modified to process the object environment related conditionals and then generate code only for the selected path.

---

I5. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language. ....T

Requirement I5

The official definition of CORAL 66 represents the kernel of the language. Blandford extension and other application oriented libraries have been added to expand the capabilities of the language.

No conflicts with other requirements.

---

I6. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition. ....P

The CORAL 66 language definition sets some limits on some of its features which are dependent only on translators. For example, it requires only one or two dimensional arrays, sets the length of the identifiers to be twelve characters (the extra characters are legal but will be ignored by the compiler) etc. However, it does not specify many other limits e.g., the number of permissible nested loops, number of nested parentheses levels in expressions, maximum number of formal and actual parameters, maximum number of identifiers in a program, etc. Hence it only partially meets the requirement.

To implement this change would require a careful analysis of user requirements and then setting a limit on these various features of the language in the official definition. Effect of such standardized limitations on existing implementations will be minimal.

---

I7. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator. ....I

CORAL 66 does not impose restrictions on the object environment. It does not state limits on storage requirements, types of peripherals, real-time clock etc. In fact, it allows language features such as the floating point to be omitted from the implementation if adequate hardware facilities are not present.

CORAL 66 should make the availability of floating point data computations mandatory for all implementations, as required by A2. If the computer does not contain the floating point hardware, the option should be supplied by the software package.

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs. ....T

No efficiency cost for unused features .....T
Efficient code can be produced for all features .....T

Efficiency in terms of code and time has been one of the major considerations in CORAL 66 design and implementation. Conceptually, CORAL 66 compilation is a one pass process. The insistence that identifiers are fully declared or specified before use simplifies the compiler by ensuring that all relevant information is available when required. The syntax of the language is transformable into one-track predictive form, which enables fast syntax analyzers with no back-tracking to be employed. Features which require elaborate storage in the object machine for efficient program execution, for example dynamic storage allocation, are not included in the language. Unless run in a special diagnostic mode, a CORAL 66 compiler is not expected to generate run-time checks on subscript bounds.

In general, CORAL 66 language features are simple, clear and concise and permit efficient code generation.

There is no potential conflict with other requirements but to meet all the requirements, definition will have to be significantly modified. In order to provide for range checking on variable subscripts (A6, D4), parameter checking on procedures (C6), variable numbers of arguments (C9), user definition of new data types and operators (B11), addition of several other data types (e.g., Boolean, status etc.) the complexity of the language will have to be increased. In some cases it is questionable if new features can be added to the existing language syntax without loss of clarity and dependent efficiency.

---

J2. Any optimizations performed by the translator will not change the effect of the program. ....T

This is implied in language definition but never explicitly stated.

---

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions. ....F

Requirement J3

In order to meet this requirement machine language insertions should only be permissible within the body of compile time variables (I4) in an encapsulated form. CORAL 66 does not provide for compile time variables. It permits easy insertion of machine code between BEGIN and END statements. Therefore, at present, machine code can be easily inserted within the body of executable statements. Hence the language does not meet this requirement.

To meet this requirement the language definition will have to define "compile time variables", their syntax and semantics and use. Procedures for defining characteristics of the hardware and insertion of machine code should be defined also within their body. The existing syntax for machine code insertions will have to be dropped. This change will impact all existing implementations.

---

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.. ....P

Encapsulated specification of representation possible .....P
Space/time tradeoff can be specified .....P

The TABLE declarations in CORAL 66 allow a programmer to control object representation of composite data. The user has the facility to specify table size, word and bit position for elements and their length, and even the bit configuration while presetting the values in these tables. The language also allows the user to pack data in tables, thus permitting him some control over space utilization. Avoidance of RECURSIVE option in programs also saves core space. Not specifying the special mode of operation which allows generation of code for run time checking of subscript ranges, the user can also control some time and space also. Other features of the language, however, do not allow the user to control space or time.

To fully meet this requirement some additional "compile time variables" can be provided directing the compiler to save (1) core or (2) time. The user should be able to specify dense, medium or rare packing of data, or avoidance of excessive code to permit faster execution time. Implementation of these optimization variables will effect existing language definition and all existing implementations.

---

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics. ....F

Open/closed properties can be specified .....F
Open and closed versions have the same semantics .....F

CORAL 66 only allows closed routines in its definition. It does not allow in-line procedures to be included in user code. Hence it does not meet this requirement.

The syntax of the procedure definition and/or the call should be modified to allow for open or closed procedures. The user will exercise his judgment as to when he wants to use open or closed procedures. It is a simple change, but will affect all existing implementations.

## Extraneous Features

We recommend that the following features or CORAL 66, not required by the Tinman, be kept:

* Parameterized MACRO capability, with nesting. This gives the functionality of open subroutines, as called for by Tinman.

* Explicit type conversions of any data unit (typed or untyped). This can be used to control the scaling on intermediate fixed point calculations (e.g., FIXED(13,5)(ABC - XYZ) specifies that the result of the subtraction is to be treated as though it had 5 fractional bits) and to override any implicit type conversions. The latter consideration would no longer be valid if the implicit type conversions were removed from the language.

We recommend that the following features of CORAL 66, not required by the Tinman, be deleted:

* Table word referencing and presetting, without consideration of the table structure. This is too hardware dependent and can seriously inhibit portability. It is also a means to defeat type checking.

* "Anonymous" referencing, which gives a capability of addressing machine words. This is too hardware dependent, defeats type checking, and its only legitimate use seems to be as a rudimentary pointer mechanism. (It is easy to think of some appalling illegitimate uses.) Presumably this feature would be unneeded if a full-blown pointer capability is present.

* Statement switches. These are elaborations of the GOTO.

CORAL 66 has one hardware feature which should probably be retained, but it should be required that each use be bracketed (encapsulated) by statements which point out the machine dependency. That feature is word logic, i.e., bit manipulation of data using the operators MASK (and), UNION (inclusive or), and DIFFER (exclusive or).

## Summary

CORAL 66 is a general purpose programming language well suited for use on small and medium sized computers. The kernel of the language provides facilities to perform fixed point computation, one or two dimensional array manipulation, short circuit evaluation of Boolean operations, mechanisms for explicit conversions of untyped expressions to integer, fixed, or floating point numbers, type checking and matching of formal and actual parameters, call by value and call by reference, literals for all language defined data types, capability of initialization of variables at the time of their definition, a 'location' option which can be used as a pointer mechanism, and fixing the scope of identifiers at compile time. CORAL 66 allows interfaces with assembly language routines. It provides for an optional definition of recursive procedures. The syntax of the language allows free formatting, is simple and permits the user to specify mnemonic names of any length. (The translator is required to check only the first eight characters to establish uniqueness.) The smallness of the language and the simplicity of its constructs allow efficient code to be generated.

In addition to the language kernel, there are some well-known extensions to the language, such as the Blandford extension. This extension provides for bit, byte, or character definitions and string manipulation. The library option of the language permits application oriented procedures to be available for use by different projects.

However, there are a number of characteristics required by the Tinman which CORAL 66 does not have. For instance, there is no provision for parallel processing, exception handling, input/output, or real-time processing. The language does not allow the user to define new data types or new operators, nor does it allow specification of operations for built-in data types. Enumeration types are not supported. Boolean type data is not supported. No provision is made in the language for specification of global precision for arithmetic computation. The floating point data type has been made optional in the language, and the exponentiation operation is not defined. The language does not require that rounding be explicit nor does it specify that the truncation must take place from the right in all computations. The language permits implicit type conversions and mixed mode arithmetic (which can be interpreted as a case of implicit type conversion). Furthermore, the language syntax does not allow specification of the ranges of variables nor are there language defined rules for operator precedence or expression evaluation. There is no provision in the language to define and store data in compools. The SWITCH instruction provided in the language does not support structured programming practices because the control does not return to the statement following the SWITCH. The SWITCH statement also does not provide for an OTHERWISE clause. The GOTO statement is unrestricted and allows transfers of control to labels outside the procedure in which the GOTO occurs. No conditional compilation facility is available to generate code for only the selected path of conditional control statements. Nor is there a facility to allow a user to specify if he wanted an open or closed implementation for his procedures.

There is no easy way to make CORAL 66 meet the Tinman requirements. To do so would require deletion and modification of existing features and the addition of several new features. The modified version of the language will look nowhere near the existing version of CORAL 66. User resistance to change will also be significant because the new language may be so distant from the existing version that in upgrading of existing programs to the new version might not be simple or even possible.

However, if the objective of upgrading the existing programs is discarded and the new language is designed to meet the Tinman requirements using selected features from CORAL 66 as a kernel (e.g. procedure definition and passing capabilities, the recursive option, FOR loop, DEFINE capability, PRESET option, explicit type conversion, explicit declarations, etc.), then it is possible to design an extensible language which has similarities to CORAL 66. The similarities would be only of a fairly general nature, however, and the new language would be quite different from CORAL 66. Such an effort would be of major proportions.

# A COMPARISON OF

# CS-4

# to

# TINMAN

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

## Introduction

This report gives a comparison of the language CS-4 to the Tinman language requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1976, Section IV). For the purposes of this comparison, CS-4 is considered to be defined by:

CS-4 Language Reference Manual and Operating System Interface
Intermetrics, Inc.
Cambridge, Mass.
October, 1975

Tinman contains 78 language requirements. This report compares CS-4 to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used. (Occasionally no text is needed if a requirement is totally satisfied.) If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - Only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

(The symbols P, P+, and P- will often be used with requirements which

are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

The report concludes with two summaries. The first is of the features of CS-4 which are extraneous to Tinman and the desirability of retaining each of them. The second is of the language as a whole and the desirability of modifying it to bring in into line with the Tinman requirements.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs. ....T

The CS-4 term for "type" is "mode". the requirement is met (p. 1).

---

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators. ....P+

Integer .....T
Floating Point .....T
Fixed Point .....P-
Boolean .....T
Character String .....T
Arrays .....T
Records .....T

Not fully met. CS-4 has no fixed point type; it has only fraction (pp. 27, 58ff.)

CS-4 does have the following types:

Integer (pp. 37-44)
Floating point (pp. 44-58)
Boolean (pp. 31-36)
Character string (pp. 100-113)
Array (pp. 82-93)
Record (called STRUCTURE; pp. 94-99)

Adding fixed point to the language would be of moderate difficulty. Designing and implementing (and explaining) the necessary scaling rules is never an easy job.

---

A3. The source language will require global (to a scope)

specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code. ....P

Global arithmetic precision specification mandatory .....F
Individual variable precision specification permitted .....T

Not fully met. Global arithmetic precision cannot be specified, and it is machine dependent (p. 204).

Precision can be specified for individual variables (pp. 44-48).

The addition of global arithmetic precision specification would not be much of a change to the language, but it would probably cost 3-6 man months per target machine to write the necessary library routines.

---

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler. ....F

Treated as exact quantities .....F
Range and step size determined at compile time .....F
Scaling handled automatically .....F

Not met; CS-4 has no fixed point.

For the scope of changes necessary to add this feature, see the evaluation of A2.

---

A5. Character sets will be treated as any other enumeration type. ....F

New sets can be defined as enumeration types .....F
ASCII and EBCDIC are provided .....F
(Conversion capability between sets is available) .....F

Not met. New character sets cannot be defined. Only ASCII is provided (p. 100). There is no conversion between character sets (obviously, because there is only one).

It would be relatively easy to add the required capability.

---

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope. ....P+

Number of dimensions is fixed at compile time .....T
Type is fixed at compile time .....T
Lower subscript bound is fixed at compile time .....F
Upper subscript bound is fixed at scope entry .....T
Subscripts only integers or from an enumeration type .....T
Subscripts will be from a contiguous range .....T

Partially met. The lower subscript bound is variable at run-time (p. 83) - a violation.

Modification to meet the requirement would be trivial.

---

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time. ....F

Alternative structures for records are possible .....N/A
Discrimination condition may be any Boolean expression .....N/A

Not met. Records do not have alternate structures (p. 87). A component may be of a discriminated union type, but strictly speaking, this is a different capability (see E6).

Meeting the requirement would cause a fairly major change, as CS-4 solves the problem in a different way.

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value. ....T

Automatically defined for any type (except...) .....T
Available for individual components .....T
(Assignment and reference via functions) .....F

Met. see pp. 25-26, 31, 39, 50, 60, 77, 89, 97, 104, 118, 163-164.

---

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity. ....P+

Partly met. There is an identity predicate (called COMPARE) for every type; it does a component by component comparison for arrays and records; and it works also for defined types (see pp. 31, 37, 50, 60, 77, 90, 97, 105, 118, 164-165).

However, the predicate does not allow comparing objects of possibly different type, except for UNION.

Change to meet the requirement would be relatively minor.

---

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration. ....P+

Built-in for all numeric and enumeration types .....T
Ordering can be inhibited when desired .....F

Partly met. See pp. 34, 42, 52, 62-63, 80. There is no way to declare unordered types.

A very simple change would be needed to provide unordered enumeration types. The PRED and SUCC functions would clearly be undefined for such types.

---

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation. ....P+

Addition .....T
Subtraction .....T
Multiplication .....T
Division with real result .....T
Exponentiation .....T
Integer and fixed point division with remainder .....P
Negation .....T

Partially met (see pp. 67-75). Violation: Integer division has no remainder (p. 72), and there is no modulus operator (pp. 41, 43).

A very easy addition.

---

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers. ....T

Never from the left for data within range .....T
Never on the right for integer and fixed point .....T
Implicit floating point rounding beyond precision allowed .....T
(Run time checks can be avoided) .....F

Met. See pp. 39, 50, 60, 67-75.

---

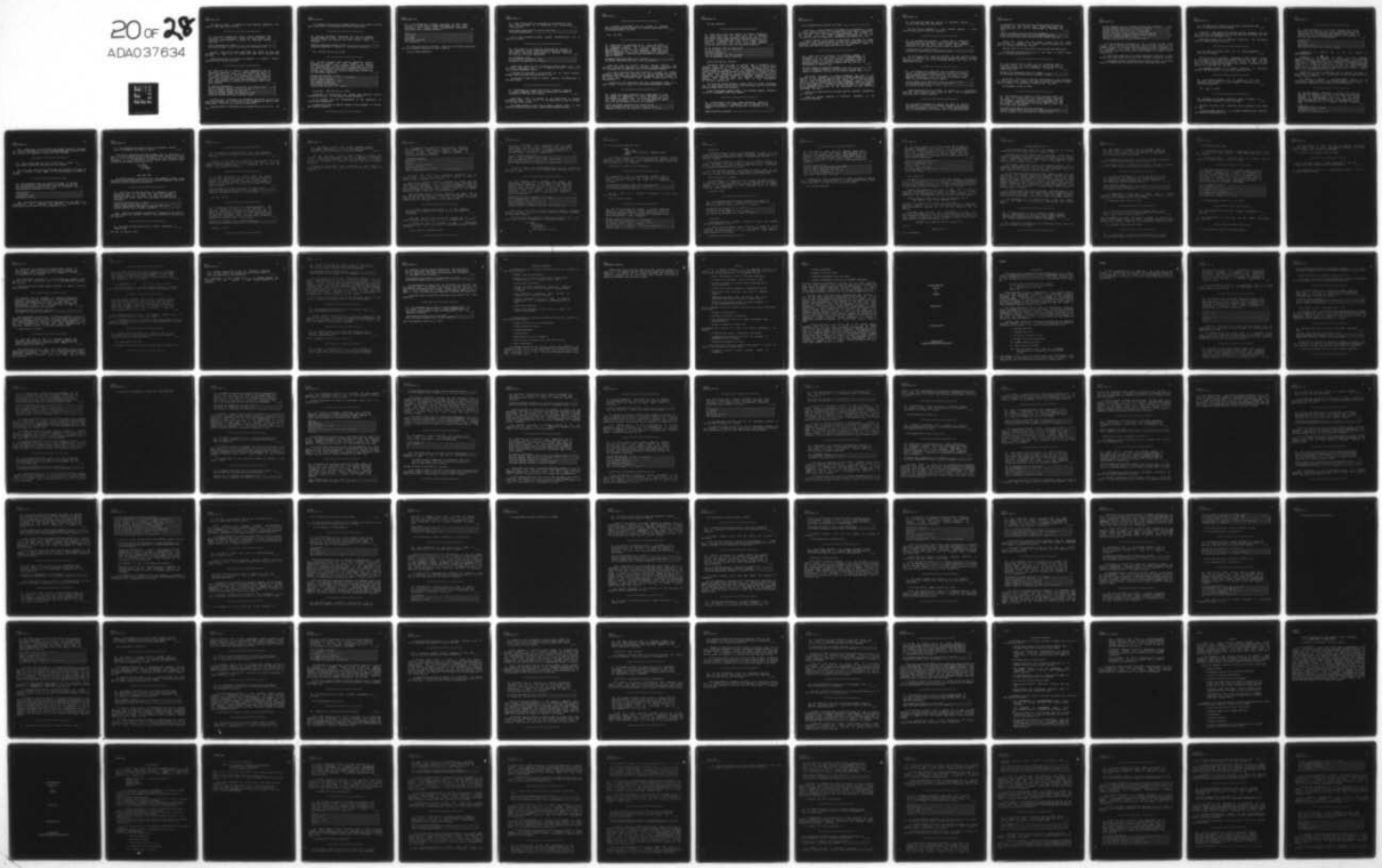
B6. The built-in Boolean operations will include "and", "or", "not", and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode. ....T

Short-circuit and .....T
Short-circuit or .....T
Not .....T
Xor .....T

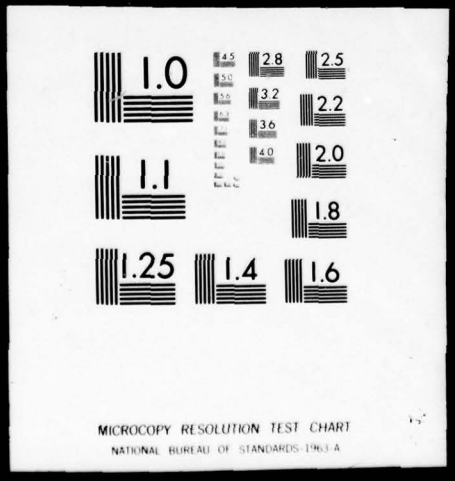
AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

20 OF 28
ADA037634



20 OF 28
ADA037634



Met (see pp. 32-33). In addition to the required operators, CS-4 also has NAND, NOR, EQV, and IMP.

---

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure. ....F

Scalar operations on arrays .....F
Assignment between records and arrays of conformable type .....F

Not met. There are no scalar operations on arrays (p. 93), and assignment is legal only if source and sink are of the same type (p. 125).

Change should not be too hard, as component by component compare operations are already available.

---

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors. ....P

No implicit conversions .....F
Explicit between integer, fixed point, and floating point .....P
Explicit between fixed point scale factors .....F
(Explicit between integer and Boolean) .....F
(Explicit between integer and enumerated types) .....T
(Explicit between different enumerated types) .....P-

Partially met. Violations: The arithmetic operations all work with mixed mode operands (pp. 71ff) which can be interpreted as implicit type conversions. There are no fixed point conversions.

Required explicit conversions are discussed on pp. 40, 51, 56, and 106.

The required change to the language would be fairly minor, but many would argue that it is unnecessary and even undesirable.

---

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated. ....T

Implicit conversion between ranges .....T
Exception condition on integer and fixed point truncation .....U

Met. See pp. 39, 50, 60, 67-75.

---

B10. The base language will provide operations allowing programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent. ....P

Sending and receiving of data .....T
Sending and receiving of control information .....F
Dynamic device assignment .....F
User exception condition control .....T
Installation independence .....U
(Data formatting capability) .....T
(Reading and writing of bit strings) .....F

Partly met. (see Part II, pp. 1-20).

Violations: (1) No provision for sending and receiving control information as such. (2) No dynamic device assignment.

It is unknown from the documentation if the capability is installation-independent.

It would be an easy to moderate change to the language to include these missing features.

---

B11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate. ....P+

Union .....T
Intersection .....T
Difference .....F
Complement .....T
Membership predicate .....T
(Set inclusion) .....T

Not fully met (see pp. 253-256). There is no difference operation; A-B can be written indirectly as NOT (A IMP B).

Trivial change necessary.

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right. ....T

Side effects must occur in left-to-right order .....T
(Embedded assignments) .....F

Met (p. 28), although we wonder whether implementations will in fact be so strict.

---

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized. ....P+

Few precedence levels .....T
No user-defined precedence levels .....T
Operands of an operation are obvious .....P

Partly met. There are only 7 precedence levels (unary + and - are of higher precedence than * or /) (see pp. 211-214). There are no user-defined precedence levels.

Violation: The operands of an operation are not always obvious, because A/B/C and A/B*C are permitted (p. 67).

It would be an easy change to require explicit parenthesizing in the above cases.

---

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed. ....P+

Partly met. There is however an odd restriction on Boolean expression forms: I > 0 AND P is not permitted; it must be written (I > 0) AND P (pp. 34-35).

The change appears so easy that we wonder whether there is some hidden difficulty arising from a specialized parsing technique.

---

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time. ....T

Met. (p. 173).

---

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to ease of learning. ....P

Parameter rules consistent in all contexts .....T
No special operations applicable only to parameters .....F

Partly met (see pp. 140-147, 150-151, 160-162, 170-172). The violation is that the operation of dynamically binding a procedure name to a procedure body is available only in the parameter context.

It is not a violation that initialization is allowed for (COPYO) procedure parameters (pp. 140-141), but not for parameters to a mode declaration (pp. 160-161), because COPYO parameters are not allowed in the latter case. Permitting fewer options in some cases is not an inconsistency.

To bring CS-4 into conformance with this requirement would be a major change. To do it right, a pointer mechanism should be added.

---

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter. ....T

Actual and formal parameters will agree in type .....T
Rank of parameter arrays is fixed at compile time .....T
Parameter array size and subscript range can be passed .....T

Met (pp. 143-147).

---

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters. ....P

Act as constants (call by value plus) .....F
Act as variables (call by reference) .....T
Exception control .....P
Procedure parameters .....T
(Act as variables, but call by value) .....T
(Act as variables, result parameter) .....T

Partly met (see pp. 139-153).

Violations: There is no way to declare that a parameter is a constant within the procedure (p. 139-151). There is a way to handle exception conditions by a sort of implicit parameter which is a "signal-handling" procedure. When an exception condition occurs, a search is made through the dynamic control chain for a procedure which can handle this exception, so such procedures are in effect implicit parameters. They may also presumably be explicit. However, there is no special format parameter class for exception handling, so strictly speaking, CS-4 does not meet this part of the requirement. (p. 151-155). See also G7.

CS-4 also has formal parameter classes which the Tinman disallows: Call by value and/or result (copy in and copy out), with no restriction against change within the procedure.

Scope of necessary change: major. The procedure passing mechanism is one of a language's most vital organs.

---

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time. ....P

Above properties optional .....F

Above properties are fixed at run time .....T

Partly met. Formal parameter attributes are not optional in CS-4; a formal parameter must be declared with a full "mode - invocation" (pp. 139-140). In the discussion of open procedures (pp. 148-149), it is implied that the formal parameter attributes can be left "unresolved", but this apparently contradicts this syntax on p. 140.

Scope of change: Fairly major, if done right. One wants to leave unspecified just those parameter attributes not needed directly in the procedure; the operations to be applied to the parameters should be specified, and this is a whole new notion of "type".

---

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time. ....P-

Variable number of arguments possible .....P-
All but a constant number of arguments have the same type .....F
Number of arguments in each call is fixed at compile time .....T

Partly met. The number of actual parameters may vary if the matching is by keyword, but the number of formal parameters is fixed. That is, if the actual parameter is not given, then a user-specified default value is assigned to the formal parameter. There is no direct way to get a count of the number of actual parameters that were supplied on a particular call. This does not meet the needs of, for example, a print procedure, so this part of the requirement is only partially satisfied. (p. 145).

There is no restriction to the same type for optional parameters. (p. 143-145).

Scope of change: Moderate to difficult, depending on the flexibility desired.

D1. The user will have the ability to associate constant values of any type with identifiers. ....T

Met (pp. 15-16), although in a very awkward fashion. A range specification must be given for constants!

---

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output). ....P

Literals for all built-in types .....T
Consistent interpretation in program and data .....U

At least partly met. There are literals for all built-in types (pp. 7-9). However, the input conversion description does not mention accuracy (Part II, p. 18), so it is unknown whether this part of the requirement is met.

---

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values. ....P+

Initial value can be specified as part of the declaration .....T
Initialization occurs at allocation scope entry .....T
No default initial values .....F

Almost completely met (pp. 18-19). An object of a user-defined type may have a default initial value (p. 166). This is technically a violation, but it seems necessary.

---

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will

be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types. ....P+

Numeric variable range specification mandatory .....T
Fixed point variable step size specification mandatory .....F
Range and step size specifications do not define a new type .....T

Partly met. Range does not define a type, and the range specification is mandatory for numeric variable (integer: pp. 37-38; real: pp. 44-46; fraction: pp. 58-60).

However, there is no fixed point, ergo no step for fixed point.

The necessary modification is part of the work of adding fixed point -- see the evaluation of A2.

---

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a contiguous subsequence of any enumeration type. ....P

Ranges of an enumeration type are allowed .....F
No arbitrary restrictions on the structure of data .....T

Partly met. There are no arbitrary restrictions on the structure of data (pp. 82-84, 94-95), but there is no way to associate a range with a variable of an enumeration (STATUS) type. (p. 76).

The necessary change is minor.

---

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables. ....F

Recursive and network structures provided .....F
Handles variable-value and structure-component connections .....F
Pointer property is an attribute of a typed variable .....F

Pointer property not for constants, affects only assignment .....N/A
Pointer property mandatory for dynamic allocation .....F
Allocation scope never wider than access scope .....U
(Either the value or the pointer is modifiable) .....F
(Pointer mechanism handles procedures and parameters) .....F
(Remap and replace assignment have different syntaxes) .....N/A
(Built-in dynamic variable creation) .....T
(Variable equivalence classes are declarable) .....F

CS-4 has no pointer mechanism as yet (p. 79-80 of Part III). Indeed, even in the section discussing a future pointer capability, the comment is made that recursive data structures are being considered as an alternative. This shows that the planning had not proceeded very far: recursive data structures alone cannot support all needed pointer capabilities.

If done right, addition of a pointer mechanism would be a major effort, and would effect such other vital features as parameter passing.

A rudimentary pointer mechanism (e.g., similar to that in PASCAL) could be patched onto the current definition with moderate effort.

E1. The user of the language will be able to define new data types and operations within programs. ....P

Partly met. New types can be defined, but new operators can not (pp. 157-172). New operations for the new types can be defined, but are accessible only via procedure calls.

Necessary changes are of moderate difficulty. The expression parcer is rather strongly affected.

---

E2. The "use" of defined types will be indistinguishable from built-in types. ....T

Partly met (pp. 17, 170-172). Components of data objects defined by the user are not accessible directly (see pp. 117, 163). The only infix operators usable with a new type are = and ~= (p. 169), but the same is true of some built-in types (e.g., STATUS). Assignment can be defined for a new type (pp. 163-164).

The necessary change is of medium difficulty. It interlocks closely with the addition of user-defined operators (E1).

---

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations. ....T

Met. (pp. 2, 14-16).

---

E4. The user will be able, within the source language, to extend existing operators to new data types. ....F

Not met. Only the = and ~= operators can be extended to new types (p. 164).

Scope of change: Moderate. It is closely connected with provision for user-defined operators (E1).

---

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented. ....P+

Construction .....T
Selection .....P
Predicates .....P+
Type conversions .....T
Operations and data can be defined together .....T

Construction is a special case of the CS-4 concept of **mode-invocation.**

Selection is accomplished by means of the standard dot-qualification. For example, if X is a variable of a defined type (mode in CS-4 terminology) and I is a "representational entity" of that type, then X.I refers to the specific instance of I contained within the "value" of X. A minor deficiency here is that all "representational entities" of a defined type are accessible in this manner. It would appear that the person who defines a type might want to hide or protect some of the "representational entities".

Predicates and type conversions can be defined by means of procedures. Predicates are not definable as special operators (e.g., infix), however (see E1).

The ability to declare a "representational entity" to be inaccessible would require a fairly minor change in the language. To allow special operators is much more complicated (see E1).

---

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time. ....T

Enumeration .....T
Cartesian products (records) .....T
Discriminated union .....T
Powerset of an enumeration type .....T

Met. Enumeration is called STATUS (pp. 76-82), Cartesian products are called STRUCTURES (pp. 94-99), discriminated union is called UNION (pp. 113-121), and powerset is called SET (pp. 253-256).

---

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired. ....T

Met. The only unions are discriminated (pp. 113-121), and there is no way to define a type (mode in CS-4 terminology) as a subset of another.

---

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type. ....T

Initialization .....T
Finalization .....T
Allocation actions .....T
Deallocation actions .....T

Met. Initialization and allocation operations can be done in a user-defined INIT procedure (pp. 165-167). Finalization and deallocation can be done in a TERM procedure (p. 167).

F1. The language will allow the user to distinguish between scope of allocation and scope of access. ....P

Partly met. Allocation and access scopes can be different, but only if the allocation scope is the whole program, similar to ALGOL 60 own (p. 20), or if the data object is shared. In the following block structure, it is not possible to have an object whose allocation scope is block B2 and whose access scope is B3.

```
B1: BEGIN
  B2: BEGIN
    B3: BEGIN
          .
          .
          .
END; END; END
```

The change should be relatively easy. The AUTOMATIC storage class declaration would simply have to carry an optional parameter, giving the block name of the allocation scope block.

---

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components. ....T

Allowable operations can be limited .....T
Access can be limited where used .....T
External declarations need not all have the same scope .....T
Naming conflicts can be avoided (renaming) .....T

Met. Limiting of allowable operations is discussed on pp. 157-172. Limiting access to external entities and renaming are discussed on pp. 174-176.

---

F3. The scope of identifiers will be wholly determined at compile time. ....T

Met (pp. 14, 126-127, 140).

---

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language. ....U

Unknown. It is implied (pp. 174-177) that the compiler will have access to entities external to the program being compiled, but it is not stated that these entities will be in libraries, and no tools for building such libraries are defined.

---

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages. ....U

Program component libraries accessible at compile time .....U
Libraries can contain foreign language routines .....U
Interface requirements checkable at compile time .....U

Unknown. See F4.

---

F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program. ....U

Libraries and compools will be indistinguishable .....U
Immediately accessible sublibraries at any level .....U

Unknown. See F4.

---

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware. ....

Not met. The CS-4 Operating System Interface provides file handling and parallel processing, but these are covered by requirements P1b and G4, not this requirement. There are no provisions for machine independent interfaces to specific peripheral equipment or special hardware.

Meeting the requirement would necessitate a major design and implementation effort, and indeed, we are not sure how one should go about it.

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling. ....P+

Sequential execution .....T
Conditional execution .....T
Iteration .....T
Recursion .....F
(Pseudo) parallel processing .....T
Exception handling .....T
Asynchronous interrupt handling .....U
Control structures from a small set of simple primitives .....F

Not fully met: CS-4 has no recursive procedures as yet (pp. 139-140). CS-4 does have interrupt handling, by means of SIGNAL-handling procedures (pp. 152-155).

The set of primitive control structures is larger and more complicated than necessary. EXIT and ABORT TO are simply euphemisms for GOTO, hence are superfluous. Worse, for parallel processing there are nine scheduling procedures, four synchronizing ones, six parameters describing priority, and eight process states. (Part II, pp. 21-34). This hardly seems a small set, or a primitive one.

Scope of change: Major. Recursive procedures are never easy to add, and they effect much else, particularly in implementation. In our view, no one has yet designed a good set of parallel processing capabilities for a language -- the problem must be hard.

---

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition. ....P

Partly met. The CS-4 GOTO can transfer control out of a scope level, although not into a scope or out of a procedure (pp. 135-136).

We regard the ABORT TO statement as only a syntactically camouflaged GOTO, and it leads out of an exception handling procedure (p. 137).

Scope of change: Relatively minor.

---

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives. ....I

Based on Boolean expression .....T
Based on type from discriminated union .....T
Based on computed choice among labeled alternatives .....T
All alternative must be accounted for .....T
Simple mechanisms will be supplied for common cases .....T

Met (pp. 122-131). Full discrimination for an IF is provided by the closing FI. The IF is a simple special-case mechanism, and it need not have an ELSE.

---

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted). ....P+

Termination can occur anywhere in the loop .....T
Multiple terminating predicates are possible .....T
Entry permitted only at the loop head .....T
Simple cases are clear and efficient .....P+
Control variable is local to the loop .....T
Control value is efficiently available after termination .....F

Partly met. The CS-4 iteration statement (REPEAT, optionally preceded by WHILE and/or FOR) is fairly clean and efficient, and meets the requirement except:

(1) There is no clean mechanism for the very common "n and a half" times loop. An EXIT, tantamount to a GOTO, must be used:

```
L: REPEAT
     {get data};
     IF {exhausted}
       THEN EXIT L
       ELSE {process data} FI;
   END
```

O.J. Dahl's syntax avoids the EXIT:

```
loop
  {get data}
  while {not exhausted} : {process data}
repeat
```

(2) CS-4 has no means of making the value of the control variable accessible after termination of the loop, except by assigning it to a variable not local to the loop on each iteration, and this is hardly the most efficient way. (pp. 132-134).

Difficulty of change: easy to moderate.

---

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure. ....F

No recursive procedures within recursive procedures .....F
(Maximum depth of recursion can be specified) .....F
(Recursiveness must be specified) .....F

Not met. CS-4 has no recursive procedures -- a major lack. (pp. 139-140).

Scope of change: major.

---

G6. The source language will provide a parallel processing capablity. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution. ....P

Able to create and terminate parallel processes .....T
Process can gain exclusive use of resources .....T
No parallel routines within recursive routines .....T
No routines within parallel routines .....F
Maximum number of simultaneous instances are declarable .....F
(Access rules are enforced) .....T

Partly met.

A parallel procedure in CS-4 is not identified as such, and may presumably also be called in-line, even recursively. There is no way to specify the maximum permissible member of simultaneous instance of a parallel process.

It is not clear when the parameters of a parallel process are bound; presumably when the process is scheduled. It is evidently not possible to define a parallel procedure within the body of a recursive or a parallel procedure (Part II, p. 24)

The CS-4 parallel processing capability, in short, while in our opinion far too complicated, satisfies the G6 requirement except that the maximum number of simultaneous activations is not declarable.

Only minor change is needed.

The capability is in direct conflict, however, with requirement G1; an operating system is required. For example, deadline scheduling implies that the CS-4 scheduling mechanism has control of the entire machine. The capabilities are even described under the heading "Operating System Interface".

Conflicts, Scope

---

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program. ....P+

Program can get control for any exception .....I
Parameters can be passed .....P
Can get out of any level of a nest of control .....I
Can handle the exception at any level of control .....I

Partly met.

Exceptions caused by "checking directives" (e.g., zero divide) carry no arguments (p. 175).

To get out of any arbitrary nest of control, an "ABORT TO label" statement is used (p. 137). This is in effect a GOTO out of a procedure, but an unavoidable one. The advantage of using ABORT TO rather than GOTO is not clear.

Relatively minor change necessary.

---

68. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation. ....P+

Priority specification .....T
Synchronization via wait/enable operations .....T
Wait for end of real time interval .....T
wait for end of simulated time interval .....F
Wait for hardware interrupt .....T
(Can enable and disable interrupts) .....T

Not fully met. There is apparently no concept of simulated time in CS-4. The TIME-WAIT procedure (Part II, p. 67) evidently causes a wait in real-time only. See also Part II, pp. 21-32, 63-67.

Minor change necessary.

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous. ....P-

Free format with statement terminator .....T
Mnemonic identifiers possible .....T
Based on conventional forms .....P
Simple grammar .....F
No special case notations .....F
No abbreviations of identifiers or keywords .....T
Unambiguous grammar .....T

Only partly met.

The syntax of CS-4 is not one of its better features. It does meet part of the requirement: It is free format with a statement terminator, it allows identifiers to be up to 32 characters long, the grammar is (to our knowledge) formally unambiguous, and there is almost no abbreviation of words (exceptions are & for AND, | for OR, and ~ for NOT).

However, the language is not easy to read. ATTR is used to introduce attributes of either formal parameters or of the whole procedure, so that parenthesis counting is often necessary to sort things out. The array declaration is very hard to read; for example:

```
ARRAY(1 THRU 2, INTEGER (RANGE:1 THRU 10) [CEIL(X)] THRU 20,
     REAL (RANGE: 0 THRU 10000, PRECISION: 10))
```

Not until '[' is reached is it clear that the INTEGER is a call on a conversion routine, not the type of the array members. It would have been much better to separate the types of the subscripts and the members.

Similarly, the mandatory range specifications on most type conversion calls make them often all but unrecognizable; see for example the conversion on page 149 of the CS-4 Manual -- it takes 11 lines! The worst example of such wordy, over-paternalistic syntax is the constant declaration; this example appears on page 16:

```
CONSTANT TWO IS INTEGER (RANGE: 2 THRU 2) [2]
```

Why not

```
CONSTANT TWO = 2
```

or, if necessary,

/

INTEGER CONSTANT TWO = 2

Hence we believe that while the CS-4 syntax may be formally unambiguous, it can be very hard for a human to read.

Moreover, the rule for blanks is very dangerous. Spaces may be inserted or omitted at will except where ambiguity arises. For example, GOTOL1 is presumably acceptable unless there is also a procedure named GOTOL1. Among other things, this makes it dangerous to add new identifiers to an existing program.

(The manual is not wholly clear on this matter of spaces. Using a different interpretation of the phrase "valid token" on page 9, one could say that GOTOL1 is not allowed, but in that case neither is GOTO -- only GO TO -- which has its own problems ).

Nor can the syntax be regarded as simple: There are some 255 syntax definitions. The initial definition of PASCAL, by contrast, has less than 100; ALGOL 60 (Revised Report), less than 125. It is not clear whether the extra complexity of CS-4 is in the language or the syntax definitions; either way, there is a problem. The CS-4 syntax is also complex enough to still contain rather basic errors; for example, it allows a procedure to be a component of an array or structure. (see pp. 29, 94). This is clearly a mistake, because there is no way to reference such a procedure.

The syntax is also not wholly consistent. Procedure parameters are enclosed in parentheses except for object-construction procedures, where square brackets are used. The CEIL, FLOOR, and ROUND conversion procedures are syntactically different from other conversion procedures.

Scope of change: Major. A redesign of many of the most complex parts of the syntax is necessary, and a clean syntax is not easy to design.

---

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences. ....I

Met. The language includes no macro facilities, and no tools for changing or adding to the syntax.

---

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset. ....F

This requirement is not met. CS-4 uses 94 printing characters, in addition to the new-line sequence and the blank (p. 5).

This requires only a relatively minor change. The only characters not in the ASCII subset are the lower case letters, {, }, and ~.

---

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals. ....F+

Break character exists .....F
(Literals are self-identifying as to type) .....T
(Bit-string literals for any type) .....F

This requirement is partly met (pp. 6-10). There is a break character (underscore) for identifiers, but not for literals. Literals for numbers and character strings are provided.

Any necessary change would be minor.

---

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings. ....F

This requirement is not met. The new-line sequence is treated as a space, which cannot appear in string and status literals, but which apparently is ignored in identifiers if no ambiguity arises (pp. 9-10).

Any necessary change would be minor.

---

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where

an identifier can be used. ....P

This requirement is partly met. In total, there are 267 key words, of which 43 are fully reserved. This seems more than a few. (pp. 261-263)

To reduce the number of keywords would be a fairly difficult change, as a syntax redesign would be required.

---

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs. ....T

Uniform comment convention .....T
Look different from code .....T
Bracketed by one or two characters .....T
Can contain any characters .....T
Can appear anywhere reasonable .....T
Terminated by the end of the line .....T
Compatible with automatic reformatting .....T

This requirement is fully met. (pp. 9-10)

---

H8. The language will not permit unmatched parentheses of any kind. ....T

This requirement is fully met; see the syntax description (pp. 209-238).

---

H9. There will be a uniform referent notation. ....P

This requirement is partly met; see the syntax description (pp. 209-238). Calls on object construction procedures use square argument brackets; all others use parentheses.

The scope of any required change is minor.

---

H10. No language defined symbols appearing in the same context will have essentially different meanings. ....T

This requirement is fully met. Verification requires a study of the entire language manual.

11. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program. ....P

This requirement is partly met. The precision for floating point arithmetic is not specifiable, and it is implementation dependent (pp. 71, 204)

Only a fairly minor change would be required to specify floating point precision.

---

12. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary. ....P+

Defaults specified for don't care cases .....P
Programmer can override the defaults .....T

This requirement is partly met. The compiler supplies defaults for standard procedures (ASSIGN, COMPARE, INIT, and TERM), for user-defined types, for procedure calls (CLOSED), for storage class (AUTOMATIC), and for checking categories (ENABLED); these can be overridden (p. 259). Only for MSTRUCTURES (pp. 189-193) must data representation be specified. However, nothing is said about reentrant code.

A minor change is needed.

---

13. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration. ....T

This requirement is met. While there are only a few built-in compile-time "variables" describing the machine configuration (e.g., MAX_MACHINE_INTEGER_VALUE, p. 204), it is possible to reference externally defined variables, which may have initial values (pp. 176-177).

---

14. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled. ....F

This requirement is not met; CS-4 has no such capability.

The change would be of small to moderate difficulty, depending on the degree of integration of the new capability with existing ones.

---

15. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language. ....F

This requirement is not met. The built-in features make CS-4 probably twice as complex as PASCAL, for example.

To restructure CS-4 as a small kernel, with rest currently built-in features provided as extension, would be a major effort, and might even change the language.

---

16. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition. ....F

This requirement is not met.

To change CS-4 to meet it would be fairly easy, but non-trivial.

---

I7. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator. ....T

This requirement is met, except that it is unknown whether the translator will give a warning if object machine capabilities are exceeded.

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs. ....P

No efficiency cost for unused features .....F
Efficient code can be produced for all features .....F

This requirement is partly met. We think that efficient code for calling exception-handling procedures will not be possible -- it is not clear why the pertinent procedure is determined by a dynamic call-chain search. We also suspect that providing for ABORT TO statements will introduce overhead not otherwise necessary, particularly when recursive procedures are added. We note that the manual does not define whether COPY0 parameters (p. 144) are set on an ABORT TO exit. The only optimization directive is NORECALL (pp. 148, 150); certainly others will be needed for really efficient code (e.g., specification of recursiveness, and of the maximum depth of recursion).

It is hard to pin down the scope of any necessary change in the absence of existing translators, but it probably is not minor.

---

J2. Any optimizations performed by the translator will not change the effect of the program. ....U

Unknown. This is a translator, not a language, requirement, and moreover the meaning of "change the effect" is itself undefined. In other words, the requirement cannot be evaluated for any language, even if it were restated with reasonable precision.

---

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions. ....T

This requirement is fully met (pp. 193-199).

---

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will

be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator. ....P-

Encapsulated specification of representation possible .....P
Space/time tradeoff can be specified .....F

This requirement is partly met. The capability is provided only for records (not, for example, for arrays as in BLISS) (pp. 190-193). Furthermore, only certain things about mapping can be specified; it takes a full left-hand-side function capability to handle all cases. There is no way to specify various densities of packing for structures.

A moderate change is required, depending on the power of the final mechanism.

---

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics. ....T

Open/closed properties can be specified .....T
Open and closed versions have the same semantics .....T

This requirement is fully met (p. 148).

## Extraneous Features

We recommend that the following features of CS-4, not required by the Tinman, be kept:

* COMPLEX type and operations.
* STRING type (assuming that the Tinman requires only a character type, not a character string type).
* Trait inquiry procedures.
* Various explicit conversions: String to Boolean, status to or from integer, string to integer, string to status.
* Real relational procedures, which perform the comparison to a specified precision.
* Various procedures: ABS, SIGN, SQRT, logarithmic, exponential, and trigonometric procedures, SUCC and PRED.
* Mixed-mode arithmetic.
* COPYI, COPYO parameter classes (call by value and result, respectively).

We recommend that the following features of CS-4, not required by the Tinman, be deleted:

* VECTOR and MATRIX types and operations.
* Cross sections of arrays.
* Keyword parameters.
* Fraction type.
* Authorities for use of features.
* Extra Boolean operators: NAND, NOR, IMP, and EQV.
* NAME parameters.

The features recommended for deletion provide only convenience, not additional power (e.g., a user can define his own MATRIX type, albeit without being able to use infix operators for it). In our judgment, the extra convenience is not worth the extra complexity, compiler size, etc., in the cases noted.

COMPLEX type is an exception; although it too could be defined by a user, it is a numeric type and the ability to use infix operators is overriding. If, however, CS-4 is extended to permit the extension of built-in operators to new data types, then COMPLEX type could be reasonably excluded from the base language.

## Summary

CS-4 is a curious language. In many important respects, it conforms to the spirit of the Tinman, and contains many features which have recently become fashionable. In particular, it has:

* Strong type-checking, with few implicit conversions.
* Powerful data structuring tools, with full safety.
* Ability to define new abstract types (although not new infix operators).
* Support for parallel processing and exception control.
* User control over precision and range of variable values.
* Complexity occurring mostly at compile time (i.e., efficient code should be possible in most cases).
* Reasonably structured control mechanisms (although it is not outstanding in this area).

It also includes most of the features necessary for systems programming. For example:

* Variable initialization.
* Attention to separate compilation.
* User control over mapping of (some) structures into physical store.
* Access to assembly language code.

In addition, the language has fairly good basic consistency. For example:

* A function can return a structure as its result.
* Expressions are acceptable in place of variables or constants in most contexts.
* Arbitrary restrictions are rare.

On the other side, CS-4 would require major change to satisfy the Tinman in the following important areas:

* Procedure parameter classes; variable number of parameters.

* Generic procedures.

* Pointers (CS-4 has none!).

* Recursive procedures (CS-4 has none).

* Machine independent interfaces to hardware components.

Of these, recursive procedures and pointers are the most important. They are basic features which have long been in some languages, and which affect (or should affect) many other vital features. It is not so clear that procedure parameters, generic procedures, and machine independent interfaces to hardware can be provided in any language in a way consistent with the spirit of the Tinman, yet efficient and clean.

Moreover, many of the more modern features have been included in what we can only regard as a pedestrian, often clumsy, way. This is clearly shown by the overall awkwardness of the CS-4 syntax (see the comments on requirement H1), the complexity of the parallel processing features, and the wordiness of declarations of new data types (even though new operators cannot yet be declared). CS-4 is also a large language, far from simple, as is clearly shown by the number of syntax definitions -- and the syntax appears to be rather carefully written. Certain features seem unnecessary; for example, the fraction type (floating point provides as much or more, in most cases), the six calling modes for procedure parameters, the system of authorities to permit a programmer to use certain language features (the compiler could instead easily print what he did use, for perusal by his manager if necessary). Another important missing capability is a system of assertions to permit the compiler to safely avoid run-time checking code.

It seems clear that if one starts with a big, rather awkward language like CS-4 and tries to modify it to meet the Tinman requirements (or any other, perhaps superior, requirements), the result will be an even bigger, clumsier, language. The requirements for simplicity, code efficiency, etc., may simply not be satisfiable given the CS-4 starting point. CS-4 does have the advantage that it doesn't really exist yet, so there are no users to insist that certain unwanted features be kept; perhaps an effort to first pare down CS-4 to a minimum, clean it up, then build it up to the requirements, might be successful. Starting with a simpler, cleaner language in the first place seems cheaper, and far less risky.

# A COMPARISON OF
# EUCLID
# to
# TINMAN

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

## Introduction

This report gives a comparison of the language EUCLID to the Tinman language requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1976, Section IV). For the purposes of this comparison, EUCLID is considered to be defined by:

> Report on the Programming Language EUCLID
> B. W. Lampson, J. J. Horning, R. L. London,
> J. G. Mitchell, and G. J. Popek
> July, 1976

Tinman contains 78 language requirements. This report compares EUCLID to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used. (Occasionally no text is needed if a requirement is totally satisfied.) If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - Only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

(The symbols F, P+, and P- will often be used with requirements which are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

The report concludes with two summaries. The first is of the features of EUCLID which are extraneous to Tinman and the desirability of retaining each of them. The second is of the language as a whole and the desirability of modifying it to bring in into line with the Tinman requirements.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs. ....T

This requirement is fully met. All identifiers must be declared before use (p. 7) and part of the declaration for data is the specification of their types.

---

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators. ....P

Integer .....T
Floating Point .....F
Fixed Point .....F
Boolean .....T
Character String .....T
Arrays .....T
Records .....T

EUCLID has no floating point or fixed point data types. They are unnecessary for the purpose of the language. The other types are all available (pp. 17-22).

The addition of these two missing types is a relatively simple task. Fixed point gives the most difficulty, in particular in defining acceptable scaling rules and in added complexity in the code generator, but neither of these is particularly difficult.

---

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code. ....F

Global arithmetic precision specification mandatory .....N/A
Individual variable precision specification permitted .....N/A

EUCLID does not have a floating point type at all.

Adding such a type to the language, with the capabilities specified in this requirement, would be a relatively simple task.

---

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler. ....F

Treated as exact quantities .....N/A
Range and step size determined at compile time .....N/A
Scaling handled automatically .....N/A

EUCLID does not have a fixed point type at all.

Adding such a type to the language, with the capabilities specified in this requirement, would be at worst of moderate difficulty. There would be a moderate impact on the code generator and the definition of acceptable scaling rules is bothersome. (In general, it is not practical to treat fixed point quantities as exact.)

---

A5. Character sets will be treated as any other enumeration type. ....F

New sets can be defined as enumeration types .....F
ASCII and EBCDIC are provided .....F
(Conversion capability between sets is available) .....N/A

EUCLID does not address the question of multiple character sets, nor is there any need to because the language does not support any input/output; the native character set of the machine is sufficient.

It would be relatively easy to add the required capability.

---

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope. ....P+

Number of dimensions is fixed at compile time .....P+
Type is fixed at compile time .....T
Lower subscript bound is fixed at compile time .....T
Upper subscript bound is fixed at scope entry .....T
Subscripts only integers or from an enumeration type .....T
Subscripts will be from a contiguous range .....T

The EUCLID definition of array type satisfies most of the features of this requirement in spirit. EUCLID allows only one-dimensional arrays, so the number of dimensions is vacuously fixed at compile time. Subscripts are from an **index type**, which is a contiguous range of either the integers or an enumeration type. The lower subscript bound is fixed at scope entry, not compile time.

In addition, the type of an array which is allocated at run-time can be a record with alternative structures, with the particular structure specified when the array is allocated. By the EUCLID use of the word **type**, the type of such an array is known at compile time, but this might not be in the spirit of this requirement.

To add multi-dimensional arrays to the language is trivial, as it is to require that the lower bound be fixed at compile time. Nor would it be much harder to rigorously require that the type of an array be known at compile time, if that is truly wanted, but this would spoil the elegant regularity of EUCLID's existing data structuring mechanisms.

---

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time. ....I

Alternative structures for records are possible .....I
Discrimination condition may be any Boolean expression .....U

This requirement appears to be fully satisfied by EUCLID. However, although the case statement has a special syntax for discriminating alternative structures, the language report makes no statement about, nor is any example given of, discriminating by means of an arbitrary Boolean expression.

If any change is necessary, it would be a very minor one.

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value. ....P+

Automatically defined for any type (except...) .....P+
Available for individual components .....I
(Assignment and reference via functions) .....F

In general, a value of any type can be assigned to a variable of that same type, subject to the EUCLID definition of identity of types. In addition, variables which have alternative record structures can be assigned a value which is one of the alternatives. The opposite assignment is not valid; the fields of the record must be assigned individually in this case. (p. 43) However, if the value is of a module type -- a generalization of the record type which may or may not include storage management -- the definition of the type must explicitly specify that assignment is an allowable operation (p. 22).

---

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity. ....P+

The equality operator is available for any two data objects of the same type, but -- as for assignment -- if that type is a module type, the definition of the type must explicitly specify that comparison is an allowable operation (see the comments on requirement B1). Comparison of objects of different types (giving a false result) is not permitted (p. 41).

The change needed to allow comparison of data of different types would be minor.

---

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration. ....P

Built-in for all numeric and enumeration types .....I
Ordering can be inhibited when desired .....F

The six relational operation are available for both numeric (integer) and enumerated types (p. 41). It is not possible to inhibit the ordering of enumerated types, however.

To allow inhibiting the ordering on enumerated types would be a trivial change.

---

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation. ....P

Addition .....T
Subtraction .....T
Multiplication .....T
Division with real result .....F
Exponentiation .....F
Integer and fixed point division with remainder .....T
Negation .....T

Exponentiation and division with a real result are not defined in EUCLID, the latter because the language does not support floating point at all. Because fixed point is not supported at all, the "integer" division operator (division with truncation) accepts only integer operands. In addition the language has a remaindering operator, mod.

Adding the missing operators to the language definition is a simple task, although the proper definition of exponentiation for fixed point operands is open to some discussion. There is a moderate to heavy impact on the code generator, particularly if two new numeric types (fixed point and floating point) are added at the same time.

---

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers. ....F

Never from the left for data within range .....F+
Never on the right for integer and fixed point .....F+

Implicit floating point rounding beyond precision allowed .....N/A
(Run time checks can be avoided) .....I

The language definition requires that the compiler verify at compile time that overflow will not occur or that it generate a legality assertion to be processed by the program verifier (p. 40). Integer data is assumed throughout the report to be exact, although no explicit statement to this effect is made, therefore precluding truncation on the right. (The result of integer operations is required to be integer -- p. 40.) The language has no fixed point type, so the question of truncation on the right does not arise. Nor does the question of rounding of floating point data arise, because that type is not supported either.

If the missing data types are added to the language, it will be a simple problem to require them to satisfy this requirement. It would be a minor change to require that code be generated instead of a legality assertion in order to check for overflow at run time. If such a feature is added to the language, it should be possible to disable the run time checks.

---

R6. The built-in Boolean operations will include "and", "or", "not", and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode. ....F

Short-circuit and .....F+
Short-circuit or .....F+
Not .....I
Xor .....F

The EUCLID report does not require that short-circuit mode of execution be used for and and or, but it contains the following statement (p. 41):

> The right operand of **and** need not be legal if the left operand is False; the right operand of **or** need not be legal if the left operand is True.

The xor operator is restricted to powersets.

There would be almost no impact to explicitly require short-circuit evaluation of **and** and **or**, because it is the easiest way to implement the above requirement. Extending **xor** to Boolean data is simple.

---

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure. ....F

Scalar operations on arrays .....F
Assignment between records and arrays of conformable type .....F

Scalar operations on arrays are not defined in EUCLID and the assignment operation is more restricted than that envisioned here (see the comments on requirement B1). Specifically, assignment between arrays and records are permitted if the assignment operands are of the same type, which means that their type definitions must be identical after certain substitutions are made. This precludes assignment between arrays and records which have the same logical structure but different physical structure.

The required capability is presumably similar to the MOVE CORRESPONDING of COBOL. This is a moderately expensive feature, although its difficulties are well-understood.

---

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors. ....F

No implicit conversions .....T
Explicit between integer, fixed point, and floating point .....N/A
Explicit between fixed point scale factors .....N/A
(Explicit between integer and Boolean) .....F
(Explicit between integer and enumerated types) .....T
(Explicit between different enumerated types) .....F

EUCLID lacks the missing conversion operations because it has no need for them. Fixed point and floating point types are not supported. Conversion between the character string form of a number and its internal form is not needed because the language has no input/output.

If the missing data types are added to the language, the numeric conversion operations can be added at the same time at modest cost. If input/output is defined, the numeric-character conversion is almost free because it is embedded in the input/output package.

---

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated. ....F

Implicit conversion between ranges .....T
Exception condition on integer and fixed point truncation .....F

The EUCLID report merely specifies that a program is illegal if a value is outside of the range of a variable to which it is assigned; it does not require that any code be generated to check for such potential truncations (p. 43). It also requires that no overflow occur during the evaluation of numeric expressions (see the comments on requirement B5).

It is a simple matter to add this requirement to the language definition and is consonant with the verifiability and legality assertion philosophy of the language. The possibility of truncation should be handled in a manner similar to the possibility of overflow (see the comments on requirement B5).

---

B10. The base language will provide operations allowing programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent. ....F

Sending and receiving of data .....F
Sending and receiving of control information .....F
Dynamic device assignment .....F
User exception condition control .....F
Installation independence .....F
(Data formatting capability) .....F
(Reading and writing of bit strings) .....F

EUCLID has no input/output capability at all.

To add an input/output capability to EUCLID, particularly of the type envisioned by this requirement, is a major task. The only simplifying aspect of that task in this case is that the designer can start from scratch, rather than having to try to embed new I/O capabilities into existing features.

---

R11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate. ....R+

Union .....I
Intersection .....I
Difference .....I
Complement .....I
Membership predicate .....I
(Set inclusion) .....I

Of the requested operations, only the set complement operator is not available in EUCLID (pp. 39-41).

It would be a minor problem to add the set complement operator to the language. The reserved word not could be used, thus not increasing the reserved word list, but other alternatives are available.

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right. ....F

Side effects must occur in left-to-right order .....F
(Embedded assignments) .....F

Because of the simplicity of EUCLID, the only possibility of side effects within an expression would be in function references. The language designers have attempted to rule out side effects entirely by severely restricting the types of formal parameters which a function can have and allowing a function to communicate with the rest of a program only through its parameters (p. 10). Unfortunately, a function can call a procedure, and the execution of the procedure can change the value of a "global" variable -- a side effect to the function call. Thus EUCLID does not satisfy this requirement.

It would be a fairly easy task to initially design a compiler to satisfy this requirement, although we consider the requirement to be questionable. The heaviest impact would be in the code optimizer, which would be restricted in the code motion it would be permitted. Since there are few EUCLID compilers in existence (perhaps none), the cost of such a change to the language specification is minimal. It is probably better, however, to change the specification to rule out side effects entirely -- i.e., to forbid procedure statements within function bodies.

---

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized. ....P+

Few precedence levels .....I
No user-defined precedence levels .....I
Operands of an operation are obvious .....P

EUCLID has six precedence levels; a fairly standard number which presumably satisfies this requirement (p. 39). There is no facility in the language for changing the defined precedence level. The usual convention of left-to-right interpretation of operators of equal precedence holds (p. 39), so that the constructs A **div** B **div** C and A **div** B * C (A/B/C and A/B*C respectively) are possible, as is A **or** B **xor** C for powersets.

To change the language to require explicit parentheses in the cases mentioned above, and other similar cases, costs little, particularly if the compiler's expression analyzer is syntax driven. The changes would

only be in the description of the syntax of expressions (which would become quite a bit more complicated through the invention of a number of new non-terminals), but would have no effect on any other part of the compiler.

---

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed. ....T

This requirement is fully met.

---

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time. ....T

This requirement is fully met (p. 32).

---

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to ease of learning. ....P

Parameter rules consistent in all contexts .....F
No special operations applicable only to parameters .....T

Only three object in EUCLID may be parameterized: procedures, functions, and types. The permissible formal parameters to functions and types are more restricted than those for procedures, in an effort to eliminate side effects. (See the comments on requirement C1.) Formal parameters are considered to be variables declared within the scope which is the procedure, function, or type body. As such they have no properties not held by any variables declared within some scope smaller than the entire program.

It would be a simple change to make the parameter rules for functions and types the same as those for procedures and would make any compiler simpler. It is doubtful that this is desirable; it would be better to change the requirement to agree with the EUCLID definition.

---

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter. ....I

Actual and formal parameters will agree in type .....I
Rank of parameter arrays is fixed at compile time .....I
Parameter array size and subscript range can be passed .....I

This requirement is fully satisfied. Actual and formal parameters must be assignment-compatible if the formal parameter is treated as a constant and they must be of the same type otherwise (pp. 43-44). An exception to the latter rule is that a formal parameter can be parameterized and the corresponding actual parameter must then be a specific instance of the formal (pp. 51-52). This implements, among other things, deferring the fixing of the specified array properties until routine invocation. The properties can be passed either explicitly or implicitly.

---

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters. ....P-

Act as constants (call by value plus) .....T
Act as variables (call by reference) .....T
Exception control .....F
Procedure parameters .....F
(Act as variables, but call by value) .....F
(Act as variables, result parameter) .....F

EUCLID permits only two classes of parameters: Those which act as constants within the routine and the classical call-by-value. There is

no class for exception control because the subject is not addressed at all by the language. Procedure parameters have probably not been considered because they are unnecessary within the design considerations of the language.

To add the missing parameter classes to the language would be a fairly major change, but not too difficult. The exception control parameter would first require that the entire question of exception handling be addressed. Then, even if they are implemented by such a simple device as a label parameter class, additional changes would be required because EUCLID has no labels at present. The implementation of procedure parameters presents no particular difficulty.

---

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time. ....F

Above properties optional .....F
Above properties are fixed at run time .....N/A

EUCLID has no generic procedure capability.

The addition of such a capability would be a fairly major change, but a well-understood one.

---

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time. ....F

Variable number of arguments possible .....F
All but a constant number of arguments have the same type .....N/A
Number of arguments in each call is fixed at compile time .....N/A

All parameterized entities in EUCLID (procedures, functions, and types) have a fixed number of arguments.

To add such a capability to the language would be a significant change. Not the least problem would be to find an acceptable syntax for

the routine declarations. Since EUCLID permits procedures to be both open and closed, and presumably these options would be available to routines with a variable number of parameters, a capability very close to a MACRO expansion would have to be added to compilers. This capability would be used for the closed case and could be used for the open case. Space considerations would probably require the definition of an elaborate linkage mechanism for the open case, however.

D1. The user will have the ability to associate constant values of any type with identifiers. ....T

This requirement is fully met. EUCLID has a constant declaration capability, and the constant can be of any built-in type, including the structure types. (p. 32)

---

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output). ....P

Literals for all built-in types .....P+
Consistent interpretation in program and data .....F

EUCLID has literals for all the "simple" built-in types, including powersets, but not for the structured types (arrays and records). The question of consistency of interpretation between program and data does not arise because EUCLID has no input/output capability.

If an I/O capability is added to EUCLID it will be a simple further addition to require consistency between programs and data but, as the Tinman infers, this can cause amazingly expensive problems in implementation.

---

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values. ....T

Initial value can be specified as part of the declaration .....T
Initialization occurs at allocation scope entry .....I
No default initial values .....T

This requirement is fully satisfied (p. 32). The language definition does not specify any default initial values, but it does not forbid any compiler to implement them.

---

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types. ....P

Numeric variable range specification mandatory .....F
Fixed point variable step size specification mandatory .....N/A
Range and step size specifications do not define a new type .....I

EUCLID permits integer variables (the only numeric types supported by the language) to be specified as one of two integer subtypes, in which case their range becomes implementation dependent (p. 18). (Ranges of integers are also possible.) All integer variables are considered to be of the same type, regardless of their range specifications.

It is a trivial change to require that ranges be specified in all cases. If fixed point type is added to the language, adding the features of this requirement to the properties of the type definitions is equally trivial.

---

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a continuous subsequence of any enumeration type. ....T

Ranges of an enumeration type are allowed .....I
No arbitrary restrictions on the structure of data .....T

This requirement is fully satisfied. In particular, any structure may contain any type of structure as a component (pp. 19-23).

---

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables. ....P

Recursive and network structures provided .....T
Handles variable-value and structure-component connections .....T
Pointer property is an attribute of a typed variable .....F
Pointer property not for constants, affects only assignment .....T
Pointer property mandatory for dynamic allocation .....T
Allocation scope never wider than access scope .....F
(Either the value or the pointer is modifiable) .....T
(Pointer mechanism handles procedures and parameters) .....F
(Remap and replace assignment have different syntaxes) .....T
(Built-in dynamic variable creation) .....T
(Variable equivalence classes are declarable) .....F

EUCLID has a pointer mechanism with the following salient features:

* Pointers point only to dynamically created variables, and they provide the only mechanism for referencing dynamically created variables.

* Part of the declaration of a pointer is specification of the **collection** to which it may point. A **collection** is a group (number undetermined) of variables of the same type. The type may be parameterized, but the parameters are either fixed at allocation time or the pointer points to a variable of a discriminated union type. Thus pointers are as type-safe as any other variable.

* The notation p^ is used to dereference the pointer p.

* The value of a pointer to a collection can be assigned to another pointer to that same collection. Thus it is possible for a pointer to point to a variable which has been destroyed.

It is impossible to estimate how much of a change is required to modify the EUCLID pointer mechanism to fulfill this requirement, because the intent of the requirement is too unclear.

E1. The user of the language will be able to define new data types and operations within programs. ....P

EUCLID possesses the generally accepted data-structuring capability, with no arbitrary restrictions (pp. 19-22). In addition, it has the **module** type, a mechanism for defining a new type and any operations peculiar to that type (pp. 22-23). The defined operations can only be procedures and functions, however, not infix operators.

The changes necessary to permit the definition of infix operators on a user-defined type would be significant. It would be necessary to change the syntax of the module definition, but the required change would not be too radical. Major changes in the lexical analyzer and syntax analyzer would be needed, however.

---

E2. The "use" of defined types will be indistinguishable from built-in types. ....T

There are no syntactic differences between built-in types and user-defined types. In particular, functions may return values of any type (p. 54).

---

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations. ....P+

Basically this requirement is satisfied by EUCLID: All identifiers must be declared. However, scattered throughout the report are a number of entities referred to as "standard" (e.g., string type and the functions abs and odd). Apparently these are intended to be standard extensions to a base definition, but in such a case it is difficult to distinguish them from part of the base language.

It would be a trivial change to satisfy this requirement by the fiat of declaring the "standard" entities to be part of the language.

---

E4. The user will be able, within the source language, to

extend existing operators to new data types. ....F

The only operations permitted on user-defined data types are in the form of procedures and function calls (pp. 22-23).

See the comments on requirement F1.

---

F5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented. ....P

Construction .....P-
Selection .....T
Predicates .....P
Type conversions .....T
Operations and data can be defined together .....T

There is no specific syntax to denoted to construction of a value of a user-defined type, but the module definition could include a function definition which could return such a value. Selection is accomplished through dot qualification: If T is a user-defined data type, C is a component of that type which has been "exported" (made accessible outside the type definition), and V is a variable of type T, then V.C refers to the instance of the component C in V. The equality predicate is pre-defined for any user-defined type, but it must be explicitly exported before it can be used; other predicates must be defined as functions or procedures. Type conversions can also be defined as functions within the type definition. (See pp. 22-23.)

Adding an explicit construction syntax to the language would cost little. Using the type identifier followed by a list of initial values enclosed in parentheses -- the syntax of a function call but with the advantage that the type name is obvious -- is one possibility (borrowed from CS-4). The appearance of such a construct would cause the execution of a particular routine defined in the type body (again, borrowed from CS-4). Such an implementation would have only a moderate impact on the syntax analyzer of a compiler and almost none elsewhere.

---

F6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian

products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time. ....T

Enumeration .....T
Cartesian products (records) .....T
Discriminated union .....T
Powerset of an enumeration type .....T

This requirement is fully satisfied (pp. 17-22, 24-25).

---

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired. ....U

EUCLID appears to satisfy this requirement. It has no overlay statement and even requires the compiler to check that machine-dependent records (in which the user specifies the bit location of all fields) do not have overlapping fields (p. 24). There are also numerous references to a "non-overlap" property. Unfortunately, there are also the concepts of **main variables, entire variables,** and **part of a variable** which we have not been able to fathom, in spite of the report's claim that they are defined precisely (pp. 33-34, 36-38). The language report, in the area where these concepts arise, raises some questions of overlapping which are not resolved.

The scope of any needed changes is impossible to determine, since it is impossible to determine what the needed changes are. Probably only a clear explanation in the report is required.

---

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type. ....T

Initialization .....T
Finalization .....T
Allocation actions .....T
Deallocation actions .....T

This requirement is fully satisfied (pp. 22-23).

F1. The language will allow the user to distinguish between scope of allocation and scope of access. ....I

EUCLID has the typical Algol-like nesting of scopes, with the declaration of an identifier in an inner scope overriding any definition in an outer scope. Thus a variable declared (existing) in an outer scope could be inaccessible in some inner scope. It is also possible to declare an identifier to be **pervasive** in some scope, which forbids another definition of that identifier in any inner scope. Thus the user can specify that the scopes of allocation and access are identical (and not necessarily the entire program). (See pr. 35-36.)

---

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components. ....P

Allowable operations can be limited .....T
Access can be limited where used .....P+
External declarations need not all have the same scope .....N/A
Naming conflicts can be avoided (renaming) .....T

Those properties of any user-defined data type which are to be accessible in the program must be explicitly declared (exported). Such properties are then fully accessible throughout the access scope of any datum of that type. Access on the user side can be limited through the device of a module; one of the properties of a module is that it forms a closed scope -- any interface with entities outside the module must be explicitly declared (they must be imported). This device satisfies the letter of the Tinman requirement, but it is probably more complicated than the authors of the Tinman envisioned. There are no declarations external to the program; the EUCLID language report does not address libraries or compools. The binding operator, ==, gives a renaming capability, among other things.

The changes necessary to add libraries to the language are discussed under requirements F4 and F5.

---

F3. The scope of identifiers will be wholly determined at compile time. ....I

This requirement is fully met (pp. 35-36).

---

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language. ....F

The EUCLID language report does not address the question of libraries.

Adding a library accessing capability to the language is a minor task. Provided that the capability is not too elaborate, its impact is usually confined to the front end of the compiler.

---

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages. ....F

Program component libraries accessible at compile time .....F
Libraries can contain foreign language routines .....F
Interface requirements checkable at compile time .....F

The EUCLID language report does not address the question of libraries.

See the comments on requirement F4. Requiring that interfaces be checked at compile time would make impact of a library facility somewhat more expensive than those comments suggest, but it could still be done for a moderate cost at most. The greatest drawback to that requirement is that it would require the relevant libraries to be accessible at compile time and would generally reduce the compiler's throughput substantially.

---

F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any

level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program. ....F

Libraries and compools will be indistinguishable .....F
Immediately accessible sublibraries at any level .....F

The EUCLID language report does not address the question of libraries.

See the comments on requirement F4.

---

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware. ....F

EUCLID has no such facility.

Adding such a facility to EUCLID, or to any language for that matter, should not be too difficult a task. The hard (and expensive) part is deciding which machine dependent capabilities are common enough and useful enough to warrant inclusion in the language. Once they have been determined, it should be fairly easy to invent a syntax for accessing these capabilities -- particularly so in a language as clean as EUCLID. The costs to the compiler are then a direct function of the elaborateness of the capability -- the number of new syntactic forms which must be analyzed, the number of new code sequences which must be generated, etc.

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling. ....I

Sequential execution .....I
Conditional execution .....I
Iteration .....I
Recursion .....I
(Pseudo) parallel processing .....F
Exception handling .....F
Asynchronous interrupt handling .....F
Control structures from a small set of simple primitives .....I

EUCLID has the usual intra-program control structures of sequential execution, conditional execution, iteration, and recursion (pp. 45-53). These structures are formed from a small set of primitives, but can be used to build quite elaborate mechanisms, particularly the looping structures with their multiple exits and parameter value generators (pp. 44, 48-50).

EUCLID has no parallel processing, exception handling, or asynchronous interrupt handling capabilities.

Adding the missing features to the language would be a major task. We do not believe that an good set of parallel processing capabilities has yet been designed, for example. The design effort, then, is a major task in itself. Assuming that whatever the final design is it will be elaborate, the impact on compilers will be substantial.

---

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition. ....F

EUCLID has neither a goto statement nor labels.

Adding this capability, as stated, to the language would be a minor problem, as the issues are well understood. Even adding a slightly more elaborate goto, permitting transfer from a scope to any containing scope, would cost about the same.

---

63. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives. ....P+

Based on Boolean expression .....T
Based on type from discriminated union .....T
Based on computed choice among labeled alternatives .....T
All alternative must be accounted for .....F
Simple mechanisms will be supplied for common cases .....T

EUCLID completely satisfies this requirement, with the exception that the use of a complementary clause (**else** after **if then** and **otherwise** after **case**) is optional (pp. 46-47).

To require the complementary clause in all cases is a trivial change to the language and any compiler, but we consider it a questionable requirement.

---

64. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted). ....P+

Termination can occur anywhere in the loop .....T
Multiple terminating predicates are possible .....T
Entry permitted only at the loop head .....T
Simple cases are clear and efficient .....P+
Control variable is local to the loop .....T
Control value is efficiently available after termination .....F

EUCLID has two iterative control structures: An "infinite loop" structure and a parameter-controlled structure (a syntactically clean version of the Algol **for**). For both of these any number of **exit** statements can be used anywhere within the loop to exit to the statement immediately after the loop. In the **for loop**, the termination test is always executed at the top of the loop. The loop parameter in the **for loop** is local to the loop and is treated as a constant within the loop body. (pp. 48-50)

Simple cases of loop termination will be handled efficiently, but they may not be clear because they require a programmed **exit** statement at the appropriate point in the loop; there are no special syntaxes for common terminating conditions other than those handled by the **for statement.** The loop control variable is not efficiently available after loop termination; it must be explicitly assigned to some variable accessible in the containing scope before exit.

The cost of adding the missing parts of this requirement to the language would be modest. For example, a simple addition to the **loop** statement would implement the common head-of-loop and tail-of-loop Boolean terminating conditions (**while** and **until**). An equally simple addition to the **exit** statement could be used to assign the loop parameter only upon exit.

---

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure. ....D

No recursive procedures within recursive procedures .....F
(Maximum depth of recursion can be specified) .....F
(Recursiveness must be specified) .....F

All routines in EUCLID are potentially recursive, without any explicit declaration. Because of this, and because a routine body can contain the definition of another routine, it is possible to define recursive procedures within a recursive procedure.

The language should be modified to require explicit specification of recursiveness. Once this is done, it would be possible to enforce the constraint against recursive procedures within recursive procedures, if this is truly desirable. In any case, run-time efficiency can be obtained in general if the compiler is not required to assume that all routines are recursive. The actual cost of such a change on compilers would be modest.

---

G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution. ....F

Able to create and terminate parallel processes .....F
Process can gain exclusive use of resources .....F
No parallel routines within recursive routines .....N/A
No routines within parallel routines .....N/A
Maximum number of simultaneous instances are declarable .....N/A
(Access rules are enforced) .....N/A

EUCLID does not support parallel processing at all.

See the comments under requirement 61.

---

67. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program. ....F

Program can get control for any exception .....F
Parameters can be passed .....N/A
Can get out of any level of a nest of control .....N/A
Can handle the exception at any level of control .....N/A

EUCLID does not support exception handling at all.

See the comments under requirement 61.

---

68. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation. ....F

Priority specification .....F
Synchronization via wait/enable operations .....F
Wait for end of real time interval .....F
Wait for end of simulated time interval .....F
Wait for hardware interrupt .....F
(Can enable and disable interrupts) .....F

EUCLID does not support parallel processing or asynchronous interrupt handling at all.

See the comments on requirement G1.

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous. ....P+

Free format with statement terminator .....T
Mnemonic identifiers possible .....T
Based on conventional forms .....T
Simple grammar .....T
No special case notations .....P+
No abbreviations of identifiers or keywords .....T
Unambiguous grammar .....T

EUCLID uses the semi-colon as a statement terminator in general and the end of a line in certain cases. Basically, the end-of-line is a statement terminator whenever the combination of the last token on the line and the next token (the report says the first token on the next line) makes it clear that the last token on the line is indeed the end of the statement. The report then gives two lists of last tokens and next tokens which specify when the implicit end-of-statement occurs. We have not verified that the concept and the specification agree. (p. 13)

The only known special case notations are the conditional escape statements: **exit when** and **return when** (p. 44). These could be written as part of an **if-then** with a negligible extra effort.

The language report contains two descriptions of the grammar: A formally ambiguous one to be read by humans and, in an appendix, an unambiguous one. We have not verified that the two describe the same language.

To remove the conditional escape statements from the language would make compilers simpler and cheaper to implement and maintain, but only slightly. The same could be said about requiring a semi-colon at the end of each statement, but the human factors considerations of making it required only when needed argue against this -- provided, of course, that the semi-colon is indeed required only when necessary and does not introduce any worse human factors problems. Unfortunately, even if the existing definition of when the semi-colon is required is "right", if EUCLID is expanded to anything close to the Tinman requirements it will probably be impossible to keep this feature in any easily understood form.

---

H2. The user will not be able to modify the source language

syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences. ....I

This requirement is fully met.

---

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset. ....F

The language character set is implementation dependent and the language designers suggest several alternatives, one of which includes the lower-case characters. Lower-case letters, when available, are considered identical to the corresponding upper-case letters, however. (p. 11)

It would be a simple matter to fix on a single character set which is contained in the ASCII 64 character subset. The language designers have suggested ways for doing this (pp. 58-59).

---

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals. ....P

Break character exists .....P
(Literals are self-identifying as to type) .....I
(Bit-string literals for any type) .....P

EUCLID has a break character for identifiers (but does not specify what it is), but not for use within literals (p. 11). Literals of different types have different syntaxes. Numeric (integer) literals may be expressed in octal and hexadecimal notation as well as decimal. There is no prohibition against continuing long character literals beyond the end of a line. (p. 14)

The blank could apparently be used as a break character in numeric literals. The cost of such a feature would be minor. It would be equally simple to require that long character literals be broken at the end of a line. This could be done by specifying that two consecutive

character literals, with possibly intervening blanks, comments, and ends-of-line, are to be concatenated into a single literal. (EUCLID already specifies a special mechanism for denoting the apostrophe -- the character string literal bracket -- inside a character string different from the common "two apostrophes mean one".)

---

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings. ....T

The language report does not specify that lexemes cannot be continued across the end of a line (other than the character string constant, see the comments on requirement H4), but the implication is strong throughout. This is probably an oversight in writing the report.

A mechanism is specified for including any object character in literal strings (p. 14).

---

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used. ....U

The EUCLID report specifies 61 "word symbols", special symbols comprised of alphabetic characters (p. 12). They are rigorously reserved; they may not be used as identifiers in any context (p. 58). However, throughout the report are references to "standard" entities, which usually have at least one identifier associated with them. The intent of these "standard" entities is unclear (see the comments on requirement F5) and, in particular, it is not clear that the identifiers are reserved. These "standard" identifiers are few enough that EUCLID has relatively few key words in any case. The only question is whether they are reserved or not.

If any change is needed, it would have a negligible cost.

---

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two)

language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs. ....P+

Uniform comment convention .....I
Look different from code .....I
Bracketed by one or two characters .....I
Can contain any characters .....P+
Can appear anywhere reasonable .....I
Terminated by the end of the line .....F
Compatible with automatic reformatting .....I

The language report specifies a single comment convention, which is to include the comments between braces ({ and }). However, implementations are permitted to realize the braces with alternate symbols, the ones specifically suggested being /* and */. Comments can contain any character other than the closing brace, can appear anywhere reasonable (not embedded in a lexeme), and are not terminated by the end of a line. They can, of course, look like code if cleverly positioned, but no more so than most comment conventions. (pp. 12, 59)

It would be a simple change to allow the end of a line to terminate a comment. This would also tend to reduce the ability to make comments look like code.

---

H8. The language will not permit unmatched parentheses of any kind. ....I

This requirement is fully met.

---

H9. There will be a uniform referent notation. ....P+

Except that functions cannot appear on the left of an assignment statement, this requirement is fully satisfied. The notation for function references and arrays is identical. In particular, arrays can have structured components, functions can return structured values, and the same notation is used to reference a component of a structured array and a component of a structured function result.

Adding the ability to assign to a function reference would be fairly expensive, and probably not worth the cost.

---

R10. No language defined symbols appearing in the same context will have essentially different meanings. ...."

A relatively large number of symbols in EUCLID have multiple meanings. For example, +, -, and * are used as powerset operators as well as numeric ones (representing union, difference, and intersection, respectively). This has some historical justification, but gives rise to the problems expressed in the text of this requirement. In addition, <= and >= denote set inclusion as well as the obvious numeric relationals, and the latter is even used for logical implication! (This last is a fluke of an "incorrect" specification of the representation of the Boolean values.)

To remove these multiple use symbols is an easy task. They should be retained for their numeric meanings and new key words invented for the other uses. The cost of such a change is negligible.

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program. ....P

EUCLID contains a relatively large number of implementation dependent defaults for such a modern language. One section of the language report (section 13) specifies minimums for some of these, but the implementation dependence is still present. Few of these defaults will affect program logic (the ranges of the signed and unsigned integer types, which will be fixed by the hardware of the object machine, are exceptions), but the dependence does mean that programs which compile correctly on one compiler may have to be modified to compile on another.

It is a simple task to specify the default values, and to eliminate the signed and unsigned integer types (requiring an explicit range specification). The problems of enforcing these specifications by assuring that no compiler exceeds them might be insurmountable, even in the relatively tightly controlled DOD environment.

---

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary. ....P+

Defaults specified for don't care cases .....P+
Programmer can override the defaults .....P+

EUCLID does provide defaults for the generally accepted "don't care" cases: Data representation, open or closed subroutine expansion (but not reentrant or nonreentrant code), etc. With most languages it is possible to find places where an override capability might be wanted and EUCLID is no exception. For example, the language report suggests that compilers should insert "small" routine bodies in line, but the programmer who disagrees with the compiler's judgment about what is small has no ability to require the routine to be closed.

The hardest part of modifying EUCLID to make it comply with this requirement is identifying all the "don't care" cases which might need to be overridden. The cost of the necessary changes cannot be determined until they are identified, but it should not be too great.

---

I3. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration. ....F

EUCLID has no such facility.

The necessary addition would be fairly easy, provided the feature is not too elaborate. See the comments on requirement I4.

---

I4. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled. ....F

EUCLID has no conditional compilation capability.

This facility can usually be implemented in most compilers at a modest cost, the cost depending primarily on the source level elaborateness of the facility. If there are any compilers which need to be changed, the changes usually can be isolated in their initial phases.

---

I5. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language. ....I

EUCLID is a small, powerful, language, as far as it goes. Furthermore, there seems to be a concept of a kernel of the language, with various entities being described as "standard", although the identifiers associated with them do not appear in any reserved word list. These entities appear to be extensions to the base language.

---

I6. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition. ....P

A number of translator dependent limits -- including some of the ones specifically mentioned in this requirement -- are not fixed by the language report. However, minimums for these limits are specified (p. 5°). It is possible to complain that at least one of these (the maximum number of elements in a powerset --16) has been set too low.

It is an easy task to fix these limits in the language definition and the effect on most compilers would not be great, but -- as for the compiler defaults (requirement I1) -- it may not be possible to control compiler writers to prevent them from treating the limits as minimums.

---

I7. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator. ....P+

This requirement is basically satisfied. It is possible to look at the low "limit" (minimum maximum) of 16 elements in a powerset (p. 58) as having been dictated by the word size of a large number of existing minicomputers.

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs. ....U

No efficiency cost for unused features .....U
Efficient code can be produced for all features .....F

EUCLID is a clean language and the designers have given quite a bit of consideration to implementation questions. There is reason to believe that a compiler can be built which would not charge the user for unused features, but there is no compiler in existence at present to verify this belief.

There are some features of EUCLID which are potentially inefficient, and these are acknowledged in the language report (pp. 37-38). Such potential inefficiencies are in almost any high-level language, and getting rid of them is impossible within current technology, at least if the language is to be useful. It is the responsibility of the user to learn the inefficiencies of the language and decide if the feature should be avoided.

---

J2. Any optimizations performed by the translator will not change the effect of the program. ....U

This is a compiler requirement, not a language requirement, and is thus not addressed by the language report.

---

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions. ....P+

A machine code capability is defined through the use of machine instructions in routines (procedures and functions). The routines can be declared with an **inline** attribute, which requires that they be expanded in line. Thus the user has a full machine code capability, but with some clumsiness involved if all he wants is to insert a few instructions into a single location in his program. (pp. 51-57)

It would be a simple job to invent a syntax which permits in-line insertions of machine code. There would be almost no cost to any compiler because the ability to recognize and compile machine code instructions is already present.

---

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator. ....P+

Encapsulated specification of representation possible .....I
Space/time tradeoff can be specified .....P

EUCLID has a machine-dependent record capability, which permits the exact specification of the positions of fields within records (p. 24). Other data structures may be specified with the packed attribute, which directs the compiler to conserve space at the expense of time (p. 19). There is no attribute which directs the compilers to generate the most time-efficient structure, however. Thus if the compiler's default data structure requires overly clumsy code for some particular use, the programmer is forced to rely on machine-dependent records, and these sometimes result in inefficient code themselves.

It is easy to invent a new attribute to further refine the types of data layouts generated by the compiler, but the expense can be moderate to high, particularly in the code generator.

---

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics. ....P

Open/closed properties can be specified .....P
Open and closed versions have the same semantics .....I

It is possible to specify that a routine is to have an open expansion and the language report specifically calls for no difference in semantics between the two types of expansion (pp. 51, 54). It is not possible to specify that the routine must have a closed expansion, however.

It is a trivial task to add a closed attribute to the routine declarations and the cost to compilers would be negligible.

## Extraneous Features

We recommend that the following features of EUCLID, not required by the Tinman, be kept:

* String type (assuming that the Tinman requires only a character type, not a character string type).
* Octal and hexadecimal representations for numeric constants. These are occasionally useful and cost very little. A binary capability might be added as well.
* The **mod** operator (remaindering).
* **Zone**, which allows the programmer to implement is own dynamic storage allocation when desired.
* The **elseif** clause of the **if** statement. This implements a common structure often overlooked in other languages.
* The **exit** statement. This is a surrogate for **goto**, but it has useful documentation attributes.
* The numeric functions **abs** and **odd**.
* The numeric-character conversion functions **chr** and **ord**.
* The successor and predecessor functions, **succ** and **pred**, for use with enumeration types.

We recommend that the following features of EUCLID, not required by the Tinman, be deleted:

* The **storageUnit** and **storageUnitBits** types. Their purpose is unclear and they are highly machine dependent.
* The **signedInt** and **unsignedInt** types. Range specification should be required of all integer data.
* Legality assertions. These conflict with the Tinman requirement that all features be within the state of the art (requirement M1).
* Specification of internal representations. These are incomplete and even "wrong". For example, **True** must be represented by binary zero and **False** by binary one, exactly the reverse of the "natural" representation on many computers.

* The requirement that fields in machine-dependent records must not overlap. To remove this requirement permits all sorts of mischief, but the ability to do strange things is one of the reasons for needing user-defined record structures.

* Restricting pointers to point to dynamically created variables. Removing this restriction gives up some protection, but that loss is offset by the increased capability.

* Dual notations for field referencing and pointer dereferencing. Multiple notations lead to lack of clarity in program listings.

In addition, EUCLID defines an unchecked type-conversion (p. 31). This feature is useful, even necessary in some cases, but it is machine-dependent. Uses of this feature should be bracketed (encapsulated) by statements which point out that dependence.

## Summary

EUCLID is a small, elegant, language designed for writing verifiable system programs. The designers have been rigorously guided by those dual goals -- verifiability and system programming -- and have produced a spare language, yet one which is quite powerful within its tightly circumscribed limits. They have also been guided by implementation considerations, with the result that EUCLID appears to be eminently implementable.

We say "appears to be" because EUCLID is at present a paper language -- there are no known compilers, although at least one is being developed. Thus there is no experience to back up any of the opinions expressed in this report, particularly those concerning efficiency.

Among the "unusual" features required by the Tinman, EUCLID has:

* Strong typing.
* A complete, regular, data definition facility.
* A lack of arbitrary restrictions.
* Recursive routines and data structures.
* Scope rules which permit the scope of allocation to be larger than the scope of access, but which can also force them to be the same, yet not the entire program.
* A useable pointer mechanism, although probably more restrictive than is desirable in the DOD environment.
* Free format of the source program with a context sensitive rule which permits omitting the statement terminator in many cases.

Because of its size and conflicts with the two design goals, EUCLID lacks the following features, required by the Tinman:

* Fixed point and floating point data types.
* Statement labels and the **goto** statement.
* Input/output.
* Exception handling.
* Parallel processing.
* An ability to define new infix operators or to extend existing operators to new data types.

* Special syntaxes for some common control structures (e.g., **loop while** and **loop until**).

* Procedure and exception parameter classes.

It is difficult to keep from being overly enthusiastic about EUCLID. The designers have produced a powerful, elegant, language. Of course, it is to be expected that problems will arise as the language is implemented, but we do not believe that they will be major. As mentioned above, many of the missing features have been left out because they conflict with the two basic design goals. I/O and fixed point and floating point types are not needed in system programming, for example. A large number of these missing features could be added to the language with little difficulty, as they present compiling problems which are well understood. Others -- parallel processing and infix operator definition, for example -- are not so well understood (at least as far as agreeing on a "best" set of features or syntax) and could cause problems. Any attempt to enlarge EUCLID to bring it up to the Tinman requirements will doubtlessly result in a clumsier language, and one at odds with EUCLID's goals. (For this reason we suggest that the verification features of EUCLID be dropped. A Tinman-sized language will have features which make verification impossible within the state of the art.) Yet it appears that, given such a solid basis, if such an enlargement were done with great care there is a good chance that an appropriate realization of the Tinman would result.

# A COMPARISON OF
# JOVIAL J3B
# to
# TINMAN

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

## Introduction

This report gives a comparison of the language JOVIAL J3B to the Tinman language requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1976, Section IV). For the purposes of this comparison, JOVIAL J3B is considered to be defined by:

> JOVIAL/J3B Language Specification Extension 2
> Softech, Inc.
> Waltham, Mass.
> Document 2744-4.2
> July 1975

Tinman contains 79 language requirements. This report compares JOVIAL J3B to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used. (Occasionally no text is needed if a requirement is totally satisfied.) If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

(The symbols P, P+, and P- will often be used with requirements which are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

The report concludes with two summaries. The first is of the features of JOVIAL J3B which are extraneous to Tinman and the desirability of retaining each of them. The second is of the language as a whole and the desirability of modifying it to bring in into line with the Tinman requirements.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs. ....I

JOVIAL/J3B requires that the type of each variable, data table, array, etc. be explicitly specified by the user at the time of data definition. All items must be declared before use. The type of operations and expressions are not specified by the user, but are determinable at compile time. The type of any item cannot be changed at run-time. Hence J3B meets this requirement. (pp. 4-3, 4-14)

No conflict with other requirements.

---

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators. ....I

Integer .....I
Floating Point .....I
Fixed Point .....I
Boolean .....I
Character String .....I
Arrays .....I
Records .....I

J3B allows signed or unsigned integers, single or double precision floating point numbers, fixed point numbers, bit strings (Boolean variables), character strings, arrays, and tables (records), as built-in data types. It therefore fully meets this requirement. (p. 1-2, Section 2).

No conflict with other requirements.

---

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic

and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code. ....P-

Global arithmetic precision specification mandatory .....F
Individual variable precision specification permitted .....?

J3B does not require that the precision for all computations to be performed in floating point arithmetic be declared globally for any scope. Only individual specification of single or double precision floating point variables is permitted. The actual precision is dependent upon the word size of the computer and its representation. The precision specification for fixed point numbers is also machine dependent and is not controlled by the user. (pp. 1-5, 1-13, 1-14, 2-9, 2-10, 2-11, 6-2)

To meet this requirement the J3B language will have to be modified to require that precision for all floating point arithmetic be declared globally. Furthermore, it should permit explicit precision specification for individual floating and fixed point variables rather than making them hardware word-size dependent as at present.

Implementation of these changes will affect all existing implementations. Expression processing, data definition, allocation procedures, and the syntax checking mechanism will have to be modified.

---

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler. ....P

Treated as exact quantities .....?
Range and step size determined at compile time .....F
Scaling handled automatically .....T

Most representations for fixed point numbers do not permit exact representation of many decimal fractions. Hence, a JOVIAL/J3B literal value may not denote its true decimal value but rather an implementation-dependent approximation to its true value in the set of <FIXED(A)> values. The range and step size for fixed value variables are not specified by the user either. Scaling is handled automatically. (p. 1-14, 2-11.)

The exact representation of a decimal number into binary is a practical impossibility for existing types of computers. However, more

visibility can be allowed the user by providing truncation and rounding routines (e.g., round up, round down, round even, etc.) in the language library. Furthermore, the language should permit user specification of the level of precision for these numbers. Range and step size should also be explicitly specified in the language syntax for fixed point variable declarations.

Implementation of these changes in the language features will have some impact on all existing compilers in the areas of syntax analysis, library routines, storage allocation, etc.

---

A5. Character sets will be treated as any other enumeration type. ....F

New sets can be defined as enumeration types .....F
ASCII and EBCDIC are provided .....F
(Conversion capability between sets is available) .....N/A

J3B does not allow the user to specify and enumerate his own character set. The user must use the standard language-defined character set consisting of 26 upper case alphabetic characters, the decimal digits, and 13 special characters. The provision for ASCII or EBCDIC code for the character set and conversion between two sets of codes is not controlled by the user and is completely hardware and installation dependent. Hence, J3B does not meet this requirement. (pp. 2-2, 2-3.)

To meet this requirement the language must provide for status variables and allow enumeration of character sets defined by the user. The library must support conversion routines between ASCII and EBCDIC which must be available to the user. The user program should then be allowed to use the user defined character set.

Implementation of these changes in the language will have a major impact on all existing implementations. The entire lexical and syntax analysis phases of the compiler (string manipulation, comments etc.) will have to be modified.

---

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will

be determinable at entry to the array allocation scope. ....F1

Number of dimensions is fixed at compile time .....T
Type is fixed at compile time .....T
Lower subscript bound is fixed at compile time .....T
Upper subscript bound is fixed at scope entry .....F
Subscripts only integers or from an enumeration type .....T
Subscripts will be from a contiguous range .....T

The J3B language specifications require that the user specify one, two or three dimensions for an array at compile time. The type of the array is fixed at compile time. All elements of the array are required to be homogeneous in type. The language automatically fixes the lower bound of the array to 0. The user must specify the upper bound of the array at compile time as a positive integer constant. The upper bound is not evaluable at scope entry time. (Section 4.4.)

To meet this requirement the language will have to allow the use of expressions in upper subscripts which evaluate to a positive integer at scope entry time. The arrays will then have to be allocated at execution time instead of at compile time as is presently done in the existing implementations.

---

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time. ....F

Alternative structures for records are possible .....T
Discrimination condition may be any Boolean expression .....F

JOVIAL/J3B provides an OVERLAY feature which allows the user to specify not only the records or variables which can overlap each other, but also the names of variables, arrays and records which are assigned successive storage locations in core. For example, OVERLAY A,B,C=FI will allow allocation of A, B, and C to be continuous, one after another, while at the same time FI will be allocated starting at the location where C was allocated. Each variable will be available to the user at run time. However, the language does not provide for a Boolean expression to be used as the discrimination condition to be used for field selection. Hence it does not meet that portion of the requirement. (Section 4.6.)

To meet this requirement the language syntax will have to be modified to allow selection of overlaid fields after evaluating a Boolean expression. The result of the evaluation will determine which of the overlaid fields is to be selected and returned to the user.

The existing compilers do not provide such facility and will have to be modified to accommodate this change in the language.

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value. ....P

Automatically defined for any type (except...) .....F
Available for individual components .....T
(Assignment and reference via functions) .....T

Assignment and reference operations are defined for the built-in data types in J3B. Values for a built-in type can be assigned to the variable, pseudo variable, array component or record component of that type. Additionally, conversions are allowed to assign values of a different type to a target of built-in data type. The assigned values can be referenced by using the name of the target. However, the assignment operation is not defined for conformable arrays and records as a whole. (Section 5.2.)

No conflict with other requirements.

---

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity. ....T

J3B provides for = sign which can be used for comparing two data objects (regardless of type) for identity. Logical and not bit by bit comparisons are made for equality. Conversions are performed before comparison as necessary. (Section 7.)

No conflict with other requirements.

---

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration. ....F-

Built-in for all numeric and enumeration types .....F
Ordering can be inhibited when desired .....F

J4I provides for all six relational operators (p. 7-12) for comparison of numeric data. However, since the language does not allow

user defined data or status variables, these operators are not defined for enumerated data or user defined data. For the same reason, no mechanism is available in the language to inhibit ordering of data.

To meet this requirement the language definition will have to be extended to provide for user definition of new data types and status variables. Then language mechanisms should be provided to enumerate data in either an ordered or unordered way.

A major modification to existing implementations will be necessary to accommodate these changes in the language definition. Type checking mechanisms, syntax analyzer, dictionary and code generation phases will be affected.

---

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation. ....P+

Addition .....T
Subtraction .....T
Multiplication .....T
Division with real result .....T
Exponentiation .....T
Integer and fixed point division with remainder .....F
Negation .....T

Of these arithmetic operations, J3B provides for all of them except integer division with remainder. (Sections 2.2.2, 7.1.)

The language syntax will have to be modified to provide for integer division with a remainder. The user should be able to specify a location where the remainder can be stored.

This change will have a minimal impact upon existing compilers in the areas of expression processing and syntax checking.

---

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the

specified precision will be allowed for floating point numbers. ....F

Never from the left for data within range .....F
Never on the right for integer and fixed point .....F
Implicit floating point rounding beyond precision allowed .....F
(Run time checks can be avoided) .....N/A

J3B does not specify rules for truncation and leaves them implementation dependent (p. 7-3). Therefore there is no guarantee that truncation will take place only on the right. Moreover, the specifications state that for fixed point computation, implicit truncation of precision bits on the right may occur (p. 7-10). Even the floating point rounding for both single precision and double precision numbers is implementation dependent. Hence the language completely fails to meet this requirement.

The language specifications should be modified to require all truncation to take place on the right and with warning to the user. The precision specification for fixed and floating point number should be allowed to the user. The language should permit implicit rounding beyond the precision specified by the user.

These changes in the language will require minimal changes in the existing compilers. Most existing implementations attempt truncations from the left. The areas affected will be data declarations, expression processing, and run-time diagnostics.

---

R6. The built-in Boolean operations will include "and", "or", "not", and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode. ....F+

Short-circuit and .....P
Short-circuit or .....P
Not .....T
Xor .....T

J3B provides for AND, OR, NOT and XOR Boolean operators. It permits but does not require, the short circuit evaluation of these Boolean operators. (p. 7-14).

To meet the requirement a minimal change in the language definition will have to be made to require short circuit evaluation of Boolean expressions. It will have minimal affect on those implementations of the language which do not evaluate Boolean expressions in short circuit mode.

---

87. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure. ....F

Scalar operations on arrays .....F
Assignment between records and arrays of conformable type .....F

The language definition does not specify that scalar operations and assignment are permissible on conformable arrays, nor does it allow data transfers between arrays and record of identical logical structure. Any such data transfers or scalar operations between conformable array and record structures are, therefore, implementation dependent. (Sections 4.3, 5.2.)

The scalar operations for addition, subtraction and logical comparison can be added to the language definition for conformable arrays. The assignment operation can be extended to allow data transfer between conformable records and arrays. But the extension of multiplication and division operations to conformable arrays may be meaningless.

Some of the existing implementations may have to be modified to support this extended definition of scalar operators for arrays and records.

---

88. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors. ....F

No implicit conversions .....F
Explicit between integer, fixed point, and floating point .....F
Explicit between fixed point scale factors .....F
(Explicit between integer and Boolean) .....F
(Explicit between integer and enumerated types) .....F
(Explicit between different enumerated types) .....F

J3B allows implicit conversions between integer, fixed point and floating point numbers and permits mixed mode arithmetic.

To provide for explicit conversions from one data type to another the language definition will have to be modified. Built-in functions or other language features will have to be provided to perform explicit conversions from one type of data to another. Similar functions will also be required to handle fixed point scale factors.

Implementation of these capabilities will affect all existing implementations. Major changes will have to be made in the areas of expression processing assignment and syntax checking.

---

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated. ....P

Implicit conversion between ranges .....F
Exception condition on integer and fixed point truncation .....I

J3B does not allow range specification for variables. It does, however, require an error condition to be raised if the value of an integer or fixed constant exceeds the value that can be stored in the computer words assigned to store that value (usually an integral number of full computer words is assigned). (Sections 2.2.3.1, 3.1.1, 3.1.4)

The language modification to provide for user specification of range of variables is necessary. Explicit conversion operations between ranges will not be required.

This language modification will have minimal impact on existing implementations.

---

B10. The base language will provide operations allowing programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent. ....F

Sending and receiving of data .....F

Sending and receiving of control information .....F
Dynamic device assignment .....F
User exception condition control .....F
Installation independence .....F
(Data formatting capability) .....F
(Reading and writing of bit strings) .....F

J3B does not provide for I/O features. Hence there are no language constructs which allow program interaction with files, channels, devices or terminals. No capability to read or write data on I/O devices is available and no static or dynamic assignment of I/O devices is permissible. The user can not control exception conditions either. All of these features are implementation-dependent.

A systematic, standardized set of I/O statements needs to be defined for the language to allow it to perform each of the functions listed in the requirement. It calls for introduction of such constructs as READ, WRITE (with or without format) OPEN, CLOSE, ON OVERFLOW etc. to accomplish this.

The existing implementations have defined their own non-standardized I/O features to perform some of these functions. Implementation of well-defined standardized features may require significant modifications to existing compilers.

---

R11. The language will provide operations on data types defined as power sets of enumeration types (see L6). These operations will include union, intersection, difference, complement, and an element predicate. ....F

Union .....F
Intersection .....F
Difference .....F
Complement .....F
Membership predicate .....F
(Set inclusion) .....F

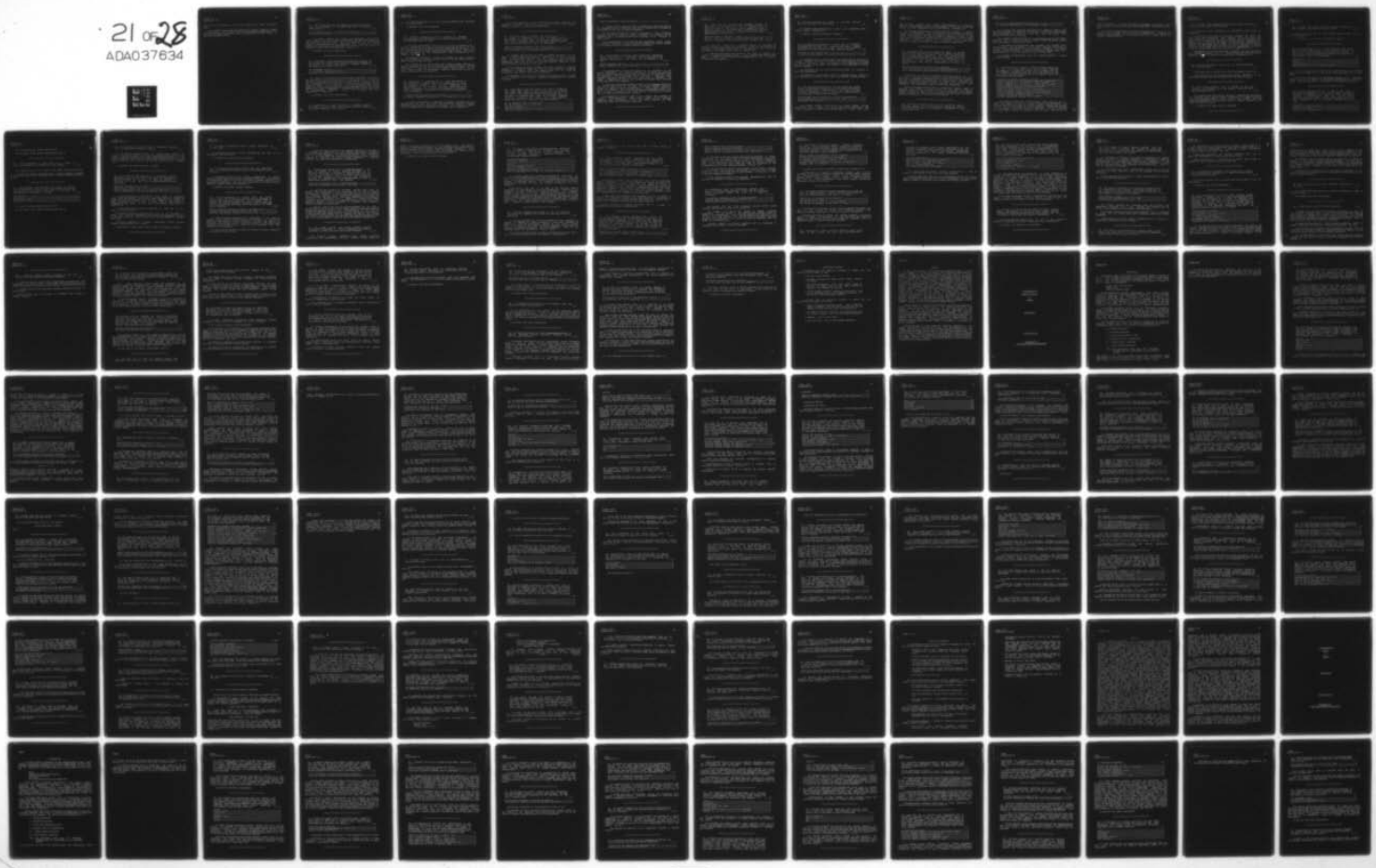
The language provides the operations of OR, AND, A-B, NOT and XOR to be performed on bit strings including the built-in data types. However the language does not provide for status variables and hence these operations cannot be applied to enumerated types or their power set. Hence the language only meets this requirement partially. (Section 7.3)

The language will have to provide a mechanism to define status variables and the enumerated types. The definition of these operations

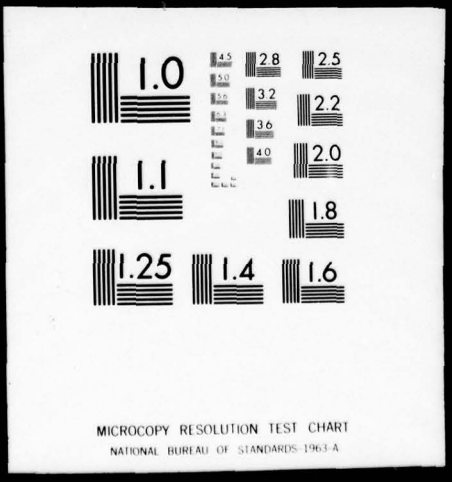
AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

21 OF 28
ADA037634



21 OF 28
ADA037634




JOVIAL J3B

will have to be extended to apply to the power set of these enumerated types.

The existing implementations will have to undergo a moderate amount of change in the areas of data definition, expression processing, IF and FOR-WHILE statement processing, etc. to accommodate the full impact of the language modification.

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right. ....T

Side effects must occur in left-to-right order .....T
(Embedded assignments) .....F

J5 provides a detailed set of rules for evaluating expressions. These rules encompass the order of evaluation for different data types and operators and also list rules for evaluation of subscripts, functions, functional modifiers, parentheses etc. In cases of operators having equal precedence, the specifications state that the order of evaluation will be from left-to-right. (Section 7)

No conflict with other requirements.

---

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized. ....T

Few precedence levels .....T
No user-defined precedence levels .....T
Operands of an operation are obvious .....T

The language establishes the hierarchy for evaluating expressions for data types to be: (1) double float (2) single float and (3) integer. The order of evaluation for operators is also defined and is as follows: (1) exponentiation (highest), (2) multiplication and division, and (3) unary and binary + and - (lowest). When two operators with equal precedence level are encountered, they are evaluated from left to right. However, the order of evaluation of constituent parts of the expression is implementation dependent. Hence, the language meets this requirements. (Section 7)

No conflict with other requirements.

---

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed. ....T

J3B allows expressions to be used wherever constants and variables of a type are allowed.

No conflict with other features.

---

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time. ....f

In the data declaration portion of the language, substitution of a constant expression in place of a constant value is not allowed in J3B. For instance, in the array declaration the dimensions of the array must be integer constants and not integer constant expressions. The same is true for constants in ITEM declarations, TABLE declarations, etc. (Section 4.3, 4.4, 4.5).

The language definition should be modified to allow constant expressions wherever constants are permitted in the data declarations or other features of the language.

This modification to the language will require minimal changes to existing compilers in the areas of lexical and syntax analysis. Storage allocation for various types of data at compile time will have to wait under after a call to expression processor which will resolve all constant expressions into constants.

---

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to ease of learning. ....P

Parameter rules consistent in all contexts .....f
No special operations applicable only to parameters .....I

J3B allows parameters for procedures, built-in operators and for declarations but does not permit type parameters, exception handling parameters and parallel processing parameters. The language however, does not permit special operations applicable only to parameters.

Language capabilities for type declaration exception handling and parallel processing will have to be added. This would require significant change to existing implementations.

---

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter. ....H+

Actual and formal parameters will agree in type .....T
Rank of parameter arrays is fixed at compile time .....T
Parameter array size and subscript range can be passed .....F

J3B requires that the type of actual and formal parameters be the same. It also requires the number of dimensions to be fixed at compile time. However, it also specifies that the values of the dimensions (i.e., the subscript range of the arrays) between actual and formal parameters agree. Thus it does not allow the size and subscript range to be passed as a parameter by the user.

If the language definition relaxed the rule requiring agreement between the values of dimensions of the array in the formal and actual parameters at compile time, then these values can be passed as parameters, thus meeting the Tinman requirement.

Modification of this language feature will require minimal changes in type checking and parameter checking mechanisms of the existing compilers.

---

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters. ....P

Act as constants (call by value plus) .....T
Act as variables (call by reference) .....T
Exception control .....F
Procedure parameters .....F
(Act as variables, but call by value) .....I

(Act as variables, result parameter) .....F

J3B allows call by value and call by reference but does not provide for exception control parameters, nor does it provide for procedures to be included in the call as actual parameters.

The J3B language definition should be extended to allow exception handling and also exception control parameters (e.g., labels) in the procedure calls. Procedure should also be allowed to be included in these calls.

These modifications to the language will constitute a major change to existing compilers. Exception handling itself will require changes or extensions to several existing language constructs.

---

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time. ....F

Above properties optional .....P-
Above properties are fixed at run time .....F

The specification of range and format is not permitted in J3B formal parameters. If a formal parameter is a constant its scale and precision can be specified by the user but there is no guarantee that the precision will be maintained by the translator. Truncation and/or rounding will occur depending upon the word size of computer. The number of dimensions as well as their values are required to be supplied by the user and is not optional.

To meet this requirement several changes in the language definition and syntax shall have to be made. The formal parameters should be allowed to optionally contain definitions of type, range, precision, scale and format. None of these options are presently allowed in an explicit form. Moreover, dynamic arrays should be permitted by allowing the dimension specification to be optional.

Implementation of dynamic arrays will require many changes to existing implementations. Other changes listed in the previous paragraph will significantly affect the syntax checking and code generation phases of the existing compilers.

---

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time. ....P-

Variable number of arguments possible .....F
All but a constant number of arguments have the same type .....F
Number of arguments in each call is fixed at compile time .....T

J3B does not provide for a variable number of arguments in procedure calls or in procedure definitions. There is no restriction on the number of arguments which must be of the same type. The number of arguments in each call must be fixed at compile time.

To meet this requirement the language will have to provide for moderate changes in language syntax. However, significant changes will have to be made in the parameter passing mechanisms of the translators to implement this change.

D1. The user will have the ability to associate constant values of any type with identifiers. ....T

The CONSTANT declaration allows a name to be associated with a constant value. (Section 4.7)

No conflict with other language features.

---

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output). ....P

Literals for all built-in types .....T
Consistent interpretation in program and data .....F

J3B provides detailed description of the properties of each type of literal (Section 1.3.3.1). The syntax for these literals is described in Section 3 in detail. There are, however, no I/O facilities described for the language nor is there a description of the relationship of input data to literals.

I/O facilities and data format description should be provided to meet this requirement.

Implementation of data format which is identical to the format of data literals will require minimal change in existing implementations.

---

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values. ....P

Initial value can be specified as part of the declaration .....T
Initialization occurs at allocation scope entry .....T
No default initial values .....F

The language permits initialization of items, tables, arrays, characters, bits, pointers etc. at the time of their definition. The actual allocation of these initial values occurs at scope entry time.

The language, however, also permits initialization of variables by default. For instance, uninitialized elements of the array or table are set to the value equivalent to all binary zeroes. (Section 4.9)

The language should forbid initialization of variables by default to meet this requirement. This would lead to two possible alternatives: either issue a warning whenever a variable, table or array is used before initialization or require that all variables, tables, arrays etc., must be explicitly initialized by the user at the time of their definition. Issuing a warning if a variable is used before being assigned a value is difficult, if not impossible in some cases.

---

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types. ....F

Numeric variable range specification mandatory .....F
Fixed point variable step size specification mandatory .....F
Range and step size specifications do not define a new type .....T

Specification of ranges of variables is not mandatory in J3B, nor is there a language option available to explicitly specify the range. The range is implicitly dependent upon the word size of the machine. The same arguments are applicable for fixed point variables. No explicit option is permitted to specify the step size of fixed point variables. Thus the language does not meet this requirement.

Explicit language options should be provided to specify range of variables and arrays. Similar options for step size specification should be provided. Implementation of these facilities will require minimal changes in the syntax checking, code generation and storage allocation functions of the existing translators.

---

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a contiguous subsequence of any enumeration type. ....F

Ranges of an enumeration type are allowed .....F
No arbitrary restrictions on the structure of data .....F

J3B does not allow explicit association of a range of values with variables, arrays, or components of a record. It does not allow defined type or enumeration of data and hence also does not permit their association with range of values.

The language restricts arrays to be declared as part of data tables (records) (p. 4-5) and vice versa. Hence it fails completely to meet this requirement. (Sections 4.3, 4.4)

To fully meet this requirement, the language will require major modifications to its definition. It must first allow user definition of new data types, enumeration of data via status variables, and permit records in array and arrays in record definitions. Then it should allow specification of a range of values for each of these cases.

All existing implementations will have a major impact as a result of these changes.

---

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables. ....P

Recursive and network structures provided .....T
Handles variable-value and structure-component connections .....T
Pointer property is an attribute of a typed variable .....F
Pointer property not for constants, affects only assignment .....T
Pointer property mandatory for dynamic allocation .....F
Allocation scope never wider than access scope .....F
(Either the value or the pointer is modifiable) .....T
(Pointer mechanism handles procedures and parameters) .....F
(Remap and replace assignment have different syntaxes) .....F
(Built-in dynamic variable creation) .....F
(Variable equivalence classes are declarable) .....F

The language provides for pointer options which can be used to share data values as well as to build recursive data structures. The pointer mechanism can be used to connect a pointer variable with a value as well as to a structure or to its component. However the language does not ensure safety while using this mechanism. It is up to the user to ensure that no values are accessed which are beyond the scope of the

pointer variable. No dynamic allocation of variables is allowed. Hence the language only partially meets the requirement. (Section 8.4, 9.7, 7.4)

To meet this requirement the pointer mechanism will have to be extended and included as part of the type definition. To provide all the capabilities listed in the requirement will be a major change in existing implementations.

E1. The user of the language will be able to define new data types and operations within programs. ....F

J3P does not permit definition of user-defined data types or operations.

The language must be changed to allow a mechanism to define new data types and operations. Once defined, these data types and operations should be given the same treatment as the built-in data types. This includes the language's ability to permit initialization, specification of actions to be taken at termination, capability to allocate and deallocate these defined data types. Defined types should be allowed to be formed from built-in types via enumeration, cartesian products, discriminated union and powerset of enumeration type.

Implementation of this definitional facility and the associated type checking required will be a major undertaking affecting syntax analysis, dictionary entries, code generation, diagnostics, etc.

---

E2. The "use" of defined types will be indistinguishable from built-in types. ....F

Since there are no defined types, this requirement is not met.

As explained in E1, the language should permit definition of new data types and make them indistinguishable from built-in types.

---

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations. ....T

J3P requires that each program component be defined and declared in the program. Items, arrays, tables, procedures, etc., must be declared at SYSTEM, COMPOOL, GLOBAL or LOCAL levels. The compiler will not substitute definitions for undeclared names by default. Instead it will give a compile-time error.

No conflict with other languages features.

---

E4. The user will be able, within the source language, to extend existing operators to new data types. ....F

Since the language does not permit defined types it does not meet this requirement.

For the scope of the needed modifications, see E1.

---

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented. ....F

Construction .....F
Selection .....F
Predicates .....F
Type conversions .....F
Operations and data can be defined together .....F

The J3B language does not provide the facility for user defined data types and declaring the list of operations applicable to that data types.

This is an extension of the facilities discussed in E1. Provision should be made to define the applicable operations for defined data types. These can be built-in operations or user-defined operations.

---

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time. ....F

Enumeration .....F
Cartesian products (records) .....F
Discriminated union .....F
Powerset of an enumeration type .....F

J3B does not support defined types at all.

For the scope of the needed modifications, see E1.

---

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired. ....F

The OVERLAY feature of J3B provides free union, among other things.

To remove the OVERLAY capability would greatly simplify existing compilers, but would have a profound effect on existing programs written in J3B.

---

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type. ....F

Initialization .....F
Finalization .....F
Allocation actions .....F
Deallocation actions .....F

J3B does not support defined types at all.

For the scope of the needed modifications, see E1.

F1. The language will allow the user to distinguish between scope of allocation and scope of access. ....1

The J3B language clearly defines and distinguishes between four types of allocation scopes: SYSTEM, COMPOOL, GLOBAL, and LOCAL. It also specifies the access scope for variables declared within each of these scopes including the rules for accessing the appropriate variable when two variables with the same name are allocated in that scope. (Section 4.2)

No conflict with other languages features.

---

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components. ....2-

Allowable operations can be limited .....F
Access can be limited where used .....P
External declarations need not all have the same scope .....F
Naming conflicts can be avoided (renaming) .....F

J3B does not allow the capability to limit access to structures where they are defined. For example, structures defined as GLOBAL are accessible from all components of a program. However, the access to these structures can be limited or where used by redefining the structure at local level. In such cases the global value will be inaccessible.

J3B does not permit a renaming capability to allow two or more names for the same entity.

Additional language capabilities will have to be introduced to specify the scope of a variable at the time of its definition and also at the time of its use. This facility must also be extended to defined data types.

A renaming capability to establish equivalence between several different names should also be added.

Introduction of these changes will affect all existing compilers.

---

F3. The scope of identifiers will be wholly determined at compile time. ....I

J3B establishes the scope of each identifier at the time of its definition. See F1 and F3.

No conflict with other language features.

---

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language. ....I

J3B provides for compools and library capabilities. It defines certain built-in functions (called functional modifiers) in the language and permits the user to add as many application oriented functions and procedures as necessary. User can also define data in various compools. Hence the language meets this requirement.

No conflict with other language features.

---

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages. ....P+

Program component libraries accessible at compile time .....I
Libraries can contain foreign language routines .....P
Interface requirements checkable at compile time .....I

J3B permits library procedures to be accessible to the user at compile time, utilizing either INLINE or COPY options. It also permits interfaces with assembly language routines, but does not specify if interfaces with compiled routines originally written in other high order languages is permissible or not. Hence it only partially meets this portion of the requirement.

J3B requires all interfaces within the library and other procedures to be checked at compile time.

To fully meet this requirement the language definition will have to be extended to specify how J3P will interact with routines written in other high order languages. The simplest way to meet this requirement will be to allow interfaces with compiled versions of these languages, requiring J3P. compilers to utilize header information in compiled programs to be used for establishing the interface.

---

F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program. ....P+

Libraries and compools will be indistinguishable .....P
Immediately accessible sublibraries at any level .....I

J3P permits both compools and libraries. Compools can be used for holding data definitions, while libraries can be used for storing procedures and functions. A project can create its own set of compools and libraries. However, the two concepts of compools and libraries are distinguishable from each other. Even their invocations in programs are different. (A declaration from a compool is accessed via COMPOOL construct (Section 3.1.2) while a library routine is accessed via a call statement (Section 5.3.1) consisting of procedure name followed by actual parameters in parentheses.)

To make the compool and library indistinguishable to the user a simple change can be made in the language syntax to make the two types of accesses identical to each other. Implementation of these changes without altering the actual implementation of compools and libraries can be accomplished with minimal effort. However, to combine the compool and library in the compilers and also make them indistinguishable to the user may require changes in compiler design of moderate proportions.

---

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware. ....I

The language provides interfaces with machine dependent capabilities via preprocessed compools, assembly language code, and

certain functional modifiers (e.g., SHIFTR, SHIFTL etc.). All machine dependent capabilities including I/O and interfaces with peripheral and special hardware must be coded in assembly language procedures. These routines can then be interfaced with J3B programs via a call to these procedures. Hence the language meets this requirement.

No conflict with other language features.

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling. ....P-

Sequential execution .....I
Conditional execution .....I
Iteration .....I
Recursion .....F
(Pseudo) parallel processing .....F
Exception handling .....F
Asynchronous interrupt handling .....F
Control structures from a small set of simple primitives .....F

The language provides for sequential execution of statements, has IF-THEN-ELSE and SWITCH conditional control statements and FOR iteration loop. It does not provide for recursion, parallel processing, exception handling or interrupt handling. It is doubtful if the existing control primitives can be used to build new control structures. (pp. 2-5, Section 5)

Major changes will have to be made in the existing control structures of the language to meet this requirement. Capabilities for recursion, parallel processing, interrupt handling and exception handling will have to be added. Furthermore, construction of complex control structures using existing control primitives requires that a new set of control primitives be developed which allows such complex control structures to be built via mechanisms such as concatenation, procedure calls, etc. All existing implementations will have a major impact on them as a result of these changes in the language.

---

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition. ....P+

The language provides for GOTO and specifies that the transfers of control be limited to labels declared in the same procedures or functions in which the GOTO occurs. However it does permit branches from a main program to the labels within its procedures or functions via procedure or function invocations and label parameters. Thus it only partially meets this requirement. (Section 5.4.1)

The language should exclude transfers of control via labels from a main program to references into its procedures and functions.

Implementation of this change will have a minimal impact on existing compilers.

---

63. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives. ....P+

Based on Boolean expression .....T
Based on type from discriminated union .....F
Based on computed choice among labeled alternatives .....T
All alternative must be accounted for .....F
Simple mechanisms will be supplied for common cases .....T

The language provides for IF-THEN-ELSE and SWITCH conditional control structures. In the IF statement, the Boolean expression (bit formula) has to be evaluated before the next course of action is determined (Section 5.5). However, it does not permit selection of an alternative based upon the value from a discriminated union. The SWITCH statement permits a choice from among the labeled variables. The otherwise clause is not provided for in the J3B SWITCH statement, hence all alternatives are not provided for in this case.

To fully meet this requirement the language has to be modified to include the OTHERWISE clause in the SWITCH statement. Furthermore it should allow selection of an alternative based upon the subtype of a value from a discriminated union in the SWITCH statement.

Minimal changes to existing compilers will be required to accommodate these language modifications.

---

64. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted). ....P+

Termination can occur anywhere in the loop .....T
Multiple termination predicates are possible .....T

Entry permitted only at the loop head .....T
Simple cases are clear and efficient .....T
Control variable is local to the loop .....T
Control value is efficiently available after termination .....P

The FOR loop in J3B very nearly meets this requirement. It allows termination of a loop via a GOTO or IF statement anywhere in the loop. Entry to the loop is permitted only at the top. The control variable is local to the loop and can be assigned to or used in formulas within the loop. But its value upon exit from the loop is dependent on the implementation and is not language defined. (Section 5.6)

The language should require that the value of the control variable should not only be available upon termination of the loop after a normal or abnormal exit, but should also be well defined. It must not be left to be implementation dependent.

Minimal changes to some of the existing implementations will be required to include this language modification.

---

65. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure. ....F

No recursive procedures within recursive procedures .....F
(Maximum depth of recursion can be specified) .....F
(Recursiveness must be specified) .....F

The language does not permit recursive options either during procedure definition or during data definition.

The language must provide the options to define recursive procedures. If the option is not specified the procedure should be taken to be non-recursive. Thus no penalty during code generation will be paid for this option. The maximum depth of allowable recursion should also be specified in the language to keep the generated code and stacks to manageable size.

Moderate changes to existing compilers will be necessary to implement this feature in the J3P language.

---

G6. The source language will provide a parallel processing capablity. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution. ....F

Able to create and terminate parallel processes .....F
Process can gain exclusive use of resources .....F
No parallel routines within recursive routines .....F
No routines within parallel routines .....F
Maximum number of simultaneous instances are declarable .....F
(Access rules are enforced) .....F

No tasking or parallel processing features are present in J3B.

Implementation of parallel processing features will require definition of language features for scheduling parallel tasks, providing for intercommunication between parallel tasks, interrupt handling, waiting and/or synchronization, program swapping, priority handling etc. This is a major undertaking which will require significant additions to existing compilers.

---

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program. ....F

Program can get control for any exception .....F
Parameters can be passed .....F
Can get out of any level of a nest of control .....F
Can handle the exception at any level of control .....F

No provision is available via the language features to provide the user with the facility to recover from arithmetic overflow, space overflow, hardware interrupts or any other exception situations.

Language facilities to provide for various exception situations should be added. Implementation of these facilities will involve a moderate to large effort in the existing compilers depending upon the number and type of exception features involved.

---

G8. There will be source language features which permit delay on any control path until some specified time or

situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation. ....F

Priority specification .....F
Synchronization via wait/enable operations .....F
Wait for end of real time interval .....F
Wait for end of simulated time interval .....F
Wait for hardware interrupt .....F
(Can enable and disable interrupts) .....F

The language does not permit priority specification by task or programs, wait option or any other real-time features.

Language definition will have to be extended to provide for various real-time features specified in the Tinman requirement. It will require moderate effort to implement these features on existing compilers.

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous. ....P+

Free format with statement terminator .....T
Mnemonic identifiers possible .....T
Based on conventional forms .....T
Simple grammar .....T
No special case notations .....T
No abbreviations of identifiers or keywords .....T
Unambiguous grammar .....P

J3B very nearly meets this requirement. It allows statements to be written in free format in columns 1-72 of a line or card, terminated by a semicolon. It allows identifiers to be two or more alphanumeric characters in length. The user can thus select mnemonic names. The compiler is obligated only to examine the first eight characters of identifiers to determine uniqueness. Many conventional notations, including the left to right precedence rules, are observed by the language. There are no language features which cater only to special cases, and no abbreviations of keywords are allowed. In most cases the language constructs are clear and unambiguous. However there are some exceptions. For instance the constructs INTGR and INTR, DEF and DEFINE are syntactically very close to each other and ambiguous.

All language primitives should be syntactically distinct from each other to avoid ambiguity. Minimal changes to the language and implementations will be required to meet this requirement.

---

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences. ....T

J3B has well-defined fixed syntax. The user is not allowed to modify any primitives or language constructs (e.g., statements).

No conflict with other language features.

---

H3. The syntax of source language programs will be corrosable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset. ....F

J3B character set consists of 26 upper case alphabets, the decimal digits, and other 18 special characters. Furthermore it allows an undefined number of implementation defined legal characters. If the implementation permits characters other than those available in ASCII, then the language permits such characters in its definition. Hence it does not meet this requirement.

J3B definition should forbid implementation defined characters, not available in ASCII character set, to be included in the language definition.

Minimal changes will be required in some implementations to provide for this language modification.

---

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals. ....P

Break character exists .....F
(Literals are self-identifying as to type) .....T
(Bit-string literals for any type) .....P

The language provides for formation rules for literals and identifiers (Sections 2.2.3, 2.2). However, the language does not define a break character for use within the identifiers.

The language should recognize a break character (e.g., underscore or space) for use within identifiers to provide readability in these constructs.

Minimal changes in syntax checking procedures will be required to accommodate this change in the language.

---

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings. ....F

J3B provides for a continuous stream of inputs. After column 72 of a card or line the first column of the next card or line is taken to be the next character in continuation. It therefore permits continuation of lexical units across lines. (pp. 2-3)

To meet this requirement the language definition will have to forbid the continuation of lexical units across lines.

Minimal changes will be required to implement this change in lexical and syntax analysis phases of compilers.

---

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used. ....I

The language allows only 53 reserved words which cannot be used as identifiers.

No conflict with other requirements.

---

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs. ....P

Uniform comment convention .....P
Look different from code .....I
Bracketed by one or two characters .....I
Can contain any characters .....P
Can appear anywhere reasonable .....P
Terminated by the end of the line .....F
Compatible with automatic reformatting .....I

The J3B language permits two comment conventions: The comments can either be enclosed between double quotes or they can be enclosed between % signs. The user can specify anything in the comments including source code which will be ignored. Semi-colons are not permitted in comments but all other characters are. Comments are permitted anywhere in the

program except in define declarations (Section 2.2.4). Comments are not terminated by the end of line; instead they are continued over the line and terminated by either the closing double quotes or the % sign. The comment conventions do not prohibit automatic reformatting of programs.

The language will have to drop one of the two sets of marks to provide for one single uniform comment convention. It should permit inclusion of all language defined characters including the semi-colon. The comments should be allowed to appear at a suitable place within the define declarations. Comments should be allowed to be terminated by the end of line.

Implementation of these changes in the comment conventions of the language will require minimal changes in existing implementations in the area of lexical analysis.

---

H8. The language will not permit unmatched parentheses of any kind. ....T

J3B requires matching, opening and closing parentheses as well as BEGINs and ENDs.

No conflict with other requirements.

---

H9. There will be a uniform referent notation. ....P+

J3B syntax for array and table reference is identical to that of its internal function calls. They all utilize parentheses after the array, table, or function name (Sections 9, 4.3 and 4.4). But whereas the arrays are permitted to appear on either the left hand or the right hand side of an assignment statement, functions are restricted only on the right hand side of the assignment statement.

However, the language also permits pseudo variables, which can appear on the left hand side of an assignment statement. (pp. 9-1)

To truly meet this requirement the language must require that all functions be capable of use as a pseudo variable and capable of being put on the left hand side of the assignment statement.

The implementation of these changes will require moderate amount of changes in the design of library functions, procedures and assignment statements.

---

H10. No language defined symbols appearing in the same context will have essentially different meanings. ....P

In general, language symbols have unique usage, but there are some exceptions. For example, the = sign is used both for assignment and as a test for equality in conditional expressions. (Sections 5.2 and 5.5)

A distinct sign for the assignment operation should be included in the language.

Minimal changes will be required to implement this change in existing compilers.

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program. ....P

Most defaults affecting program logic are specified in the reference manual with few exceptions. For example, the SWITCH statement does not contain the OTHERWISE clause (Section 5.4.3). Hence the implementors decide where the control will be transferred if the value of the index exceeds the number of labels present in the SWITCH. Similarly, since no exception control statements are provided in the language, the implementation determines what to do or where to transfer the control after an overflow or a hardware interrupt.

The SWITCH statement should be modified to provide for OTHERWISE or OUT clause. Exception control statements should be added to the language to allow the user to control the flow of program in the event of an interrupt. A moderate amount of effort will be required to extend the existing implementations to include these capabilities.

---

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary. ....P

Defaults specified for don't care cases .....I
Programmer can override the defaults .....F

The language provides for a large number of defaults to be handled by implementations. Some of these are don't care type of defaults e.g., representation of data, treating a routine as CLOSED if INLINE option is not specified, and treating a procedure as non-reentrant if the user does not specify the RENT option. Other defaults, such as the ones listed in I1 are not "don't care" type of defaults. The programmer does not have the facility to override those types of defaults.

For the scope of the needed modifications, see I1.

---

I3. The user will be able to associate compile time variables with programs. These will include variables which

specify the object computer model and other aspects of the object machine configuration. ....F

The language does not provide for selective compilation depending upon the value of certain control variables, as specified in the requirement.

To fully provide for conditional compilation facility, the user should be able to specify the name and configuration of object computer including its memory size and special hardware features. These should be taken into account during the code generation process by the compiler.

Moderate changes, including where to specify these variables in the programs, their syntax etc., and the type of code to be generated for each option will have to be made in the existing compilers.

---

I4. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled. ....T

J3P permits conditional compilation using conditional control structures. The syntax of IF statement in J3P is as follows+

IF bit formula; statement [ELSE statement];

If bit formula is a bit constant or a bit constant relational whose value is not all zero, then the statement following the semicolon is compiled, otherwise not. If an ELSE is present, only one of the statements preceding and following the ELSE is compiled. The statement preceding the ELSE is compiled when the bit constant or bit constant relational value is not all zero bits; otherwise the statement following the ELSE is compiled. (pp. 5-20)

Not compiling a statement means that syntax checking is performed but code is not generated for that statement.

No conflict with other requirements. This facility can be extended to SWITCH statement also for constant values of the index.

---

I5. The source language will contain a single clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language. ....P+

The J3B language specifications represent the kernel of the language. There are no two features which duplicate each other's capabilities with the possible exception of GLOBAL and COMPOOL scope definitions for variables. A variable can be either placed in a COMPOOL or have the GLOBAL scope, and both options will have the same effect. Hence the language nearly meets this requirement.

A redefinition of capabilities of COMPOOL and GLOBAL scopes can elimirate redundancies in the language.

Minimal changes will be required to implement these modifications in the existing compilers.

---

I6. Language restrictions which are dependent only on the translator and not on the object machine will be specified These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be explicitly in the language definition. ....P

The language specifications do not specify the maximum level of nesting in loops, the maximum number of variables that can be defined, the maximum number of subroutines allowed, number of nested parentheses levels in expressions, etc. It does, however, put a limit on the size of the variable name (8 alphanumeric characters), number of array dimensions (3) etc. (Sections 2.2, 4.4). Thus it partially meets this requirement.

The specifications should set upper limits on several factors listed above rather than leaving them implementation dependent. These limits should be reasonably high.

Implementation of these language specified limits will require minimal changes in existing compilers.

---

I7. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator. ....T

J3B language does not contain features which are dependent upon object time environment (e.g., core size, number of peripheral devices, etc.)

No conflict with other requirements.

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs. ....I

No efficiency cost for unused features .....I
Efficient code can be produced for all features .....I

The language design is such that there is no overhead in terms of generated code because of unused features. For example, there is no overhead for recursion if it is not specified by the user, no overhead for the INLINE option, etc.

No conflict with other requirements.

---

J2. Any optimizations performed by the translator will not change the effect of the program. ....I

The J3B specifications do not specify if the translator should perform optimization and how. However, the intent of the requirement is met in language specifications. Certain language features, such as INLINE, conditional IF for compilation, etc. do permit optimization of code or time and it is intended that translators should not change their effort.

No conflict with other requirements.

---

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions. ....P+

J3B requires that machine code be encapsulated inside procedures which can then be called from J3B procedures or programs via normal procedure invocation methods. Machine code cannot be interspersed inline with J3B code. However, the language does not require the assembly language procedures to have a preamble specifying such details as the name of the computer, computer word size, etc. This can, however, be accomplished voluntarily by the programmer via appropriate comments.

A mandatory procedure should be established requiring machine language procedures to specify the name, size, word size and other

details of the hardware environment. This will provide readability of machine dependent routines and make their encapsulation complete.

Minimal changes in existing implementation will be required to incorporate this language modification. The entire preamble can be treated like a comment.

---

J4. It will be possible within the source language to specify the object presentation of composite data structures. able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator. ....P

Encapsulated specification of representation possible .....P
Space/time tradeoff can be specified .....P

J3B allows a data packing factor to be specified in its table declarations which permits a user to have no packing (N), medium packing (M) or dense packing (D) of data (Section 4.3). However, no other explicit option is available to provide the translator with the specifics of data representation.

Certain other language features allow the user to control the space/time trade off. These include INLINE option (which can save time at the expense of space), RENT option in procedures (not specifying this option will save core space by generating non reentrant code) and IF conditional compile (which saves both time and space). However, no language defined directive is available which instructs the translator to consistently generate code to save time or space whenever and wherever possible throughout the process of compilation.

Language defined directives to the translator should be introduced which can specify certain formats for data before it is transferred to an I/O device or another computer. These directives should also specify whether the user wishes to have code generated which will attempt to save execution time or core space wherever possible.

Implementation of these capabilities will require modifications to existing compilers to accept language specified directives, in the area of I/O to be able to accept or transmit data in specified formats and in the areas of code optimization.

---

J5. The programmer will be able to specify whether calls on

a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics. ....T

Open/closed properties can be specified .....T
Open and closed versions have the same semantics .....T

The INLINE attribute permits a library procedure to be included in the user code at the point of invocation. If this option is not specified then the procedures are treated as closed.

No conflict with other requirements.

## Extraneous Features

We recommend that the following features of JOVIAL J3B, not required by the Tinman, be kept:

* The use of bit strings.
* Compiler directives for table packing (none, medium, or dense).
* LIKE table definition. This same effect could be obtained by other means if J3B were extended to include a type definition facility.
* Certain pseudo variables (functions) which can be used on the left hand side of an assignment statement. For example, the NEXT functional modifier.

We recommend that the following features of JOVIAL J3B, not required by the Tinman, be deleted:

* Double precision floating point. This capability would be assumed by the precision specification in any extension of the language along the Tinman lines.
* The NWDSEN functional modifier, which returns the size of a table in words. This is too hardware dependent.
* OVERLAY, a form of free union.
* Byte type data. This is too hardware dependent.

## Summary

J3P is a small language with many modern, useful features. Clarity and consistency of language feature has been emphasized throughout the language definition. The language provides for four levels of scopes (SYSTEM, COMPOOL, GLOBAL, and LOCAL) and specifies rules for resolution of conflicts related to levels of scopes of variables, data, procedures, etc. A COPY option is provided as a compile time facility to insert program text from external sources. The language also emphasizes compile-time checks, whenever and wherever possible, thus providing for debugging in early stages of program development. It requires type checking on procedure parameters and distinguishes between input and output parameters. It requires evaluation of constant expressions at compile time. Declaration for all variables, tables, records, and arrays have to be explicit and should be before use. A pointer mechanism is provided in the language. An uncommon feature in the language is conditional compilation of IF statement, in which the constant condition is evaluated at compile time and code is generated only for that portion of the condition which will be executed during execution time. A capability for specifying that a procedure is to be compiled in open form is also available (INLINE option).

However, there are several characteristics in which the language is lacking and fails to measure up to the Tinman requirements. The language does not provide for definition of new data types or new operators. It does not provide for features supporting real-time processing, parallel processing, or exception handling. No provision has been made for input/output, dynamic array allocation, truncation of results from the right instead of from the left, or explicit instead of implicit data conversion. Range and precision specifications for variables are not supported. The user does not have the facility to define procedures with a variable number of parameters. Procedures may not be recursive.

Despite the lack of many essential qualities specified by the Tinman requirements, J3P contains many fine characteristics. The existing kernel of the language is small and contains features which can be discarded, modified, or expanded, to meet the Tinman requirement. To add the missing capabilities to the language definition and to implement them are tasks of major proportions, but they have a reasonable chance of success. However, such efforts cannot guarantee a clean, consistent, and simple language. Starting from scratch would be a better approach to obtain these characteristics.

# A COMPARISON OF
# JOVIAL J73 (Level I)
# to
# TINMAN

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

## Introduction

This report gives a comparison of the language JOVIAL J73 (Level I) to the Tinman language requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1976, Section IV). For the purposes of this comparison, JOVIAL J73 (Level I) is considered to be defined by:

JOVIAL J73/1 Specification
July, 1976
Rome Air Defense Center

Tinman contains 78 language requirements. This report compares JOVIAL J73 (Level I) to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used. (Occasionally no text is needed if a requirement is totally satisfied.) If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - Only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

(The symbols P, P+, and P- will often be used with requirements which are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

The report concludes with two summaries. The first is of the features of JOVIAL J73 (Level I) which are extraneous to Tinman and the desirability of retaining each of them. The second is of the language as a whole and the desirability of modifying it to bring in into line with the Tinman requirements.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs. ....P

While J73 is typed in the sense that types are known at compile time, certain language features allow the defeat of type checking against which the compiler cannot protect. The constructs OVERLAY, based variables, and specified tables essentially implement free unions, and since there is no requirement (nor, indeed, any language artifact) to discriminate among the union types, the full intent of A1 is not satisfied in J73. (Sections 1 and 2)

The elimination of OVERLAY is trivial and has no impact on other features; the effect upon implementations is not known, but where OVERLAY is used (especially absolute overlays for system allocation), its elimination could be painful to circumvent.

Based variables could be eliminated or changed to conform to Tinman "pointer property" specifications -- decidedly expensive (see D6).

Specified tables could be eliminated, but this impacts J4, and would severely impact the extant J73/I compilers (namely, DEC-10, HBC, and SAMSO Fault-Tolerant J73's) which make heavy use of specified tables to optimize table space usage.

---

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators. ....P

Integer .....I
Floating Point .....I
Fixed Point .....F
Boolean .....P
Character String .....I
Arrays .....P
Records .....P

Boolean -- There is no Boolean type as such in J73. However, the effect of Boolean is implemented through the use of type "Bit" in which

the low-order bit of a bit string is treated as "true" if a 1 and "false" if 0. Thus, for example, predicates are evaluated to type bit of size one rather than to the conventional type Boolean.

Arrays and Records -- J73 permits both arrays (called TABLEs) and records (also called TABLEs). However, while one can declare an array of records (that's exactly what a TABLE is), one cannot declare records containing arrays. Furthermore, J73 does not support the notion of prototype definitions (MODE in ALGOL 68 or "type" in PASCAL), such that "type generators" as called for in A2 are not fully supported.

The addition of records of arrays and prototype definitions to J73 is a major extension and would greatly effect existing compiler design. Since type generation is unknown to J73, existing compilers assume a fixed number of pre-defined types in the symbol table design, and generalizing type definition is therefore a fundamental and costly change. Syntactically, one can see that the addition of prototypes is somewhat clumsy owing to the fact that J73 declarations contain property descriptions that are scattered about the declaration. To some degree, parameterized defines may be used to effect prototype definitions. Records of arrays are more difficult because of the unique naming conventions of J73 and the lack of provision for field selection by qualification.

---

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code. ....P

Global arithmetic precision specification mandatory .....F
Individual variable precision specification permitted .....T

Floating point precision is specified in the type declaration for individual variables by stating the mantissa size in bits. Thus,

ITEM FF F 27

declares a floating point variable requiring a minimum of 27 bits (excluding sign) to express the mantissa. The precision may be omitted, and the compiler supplies a default "single-precision" value that is target-machine dependent. (2.1.3.2)

A global floating point specification would affect only those variables with the default precision. The implementation impact is trivial.

---

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler. ....F

Treated as exact quantities .....N/A
Range and step size determined at compile time .....N/A
Scaling handled automatically .....N/A

J73 does not support fixed point in level I. (Sections 1 and 2)

The full language J73 includes fixed point, and presumably the exclusion of fixed point from level I reflects a concern for implementation cost only. Clearly, the full language fixed-point capability could be drawn into a level I.5 subset without a question of language consistency. The impact on existing compilers may be expected to be moderate.

---

A5. Character sets will be treated as any other enumeration type. ....F

New sets can be defined as enumeration types .....F
ASCII and EBCDIC are provided .....F
(Conversion capability between sets is available) .....F

J73 defines the character type as a built-in type. The J73 character type is really a character string type whose size is fixed at the declaration. the admissible literal denotations for character type are drawn from a large set of graphics whose ordering is implementation-dependent. (1.7.1.2, 2.1.3.2)

The addition of enumerated character types may be seen as an extension of the J73 status list (the only form of enumeration type in the language). ASCII and EBCDIC could then be added as predefined enumerations, and conversions could be added, all at modest impact on existing implementations.

---

A6. The language will require user specification of the number of dimensions, the range of subscript values for each

dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope. ....P

Number of dimensions is fixed at compile time .....I
Type is fixed at compile time .....I
Lower subscript bound is fixed at compile time .....I
Upper subscript bound is fixed at scope entry .....P
Subscripts only integers or from an enumeration type .....P
Subscripts will be from a contiguous range .....I

All array subscripts are fixed at compile time. Subscripts are confined to integer or status type (which are otherwise treated as integers). J73 does not allow character types as subscripts (except in an explicit type conversion to integer which is the character's internal bit representation), and since Tinman requires that character type be enumerated, J73 partially fails this requirement. (2.1.5)

Adding variable upper bounds to arrays will affect existing implementations in which data allocation is fixed at compile time -- moderate input. Adding character types as subscripts requires the implementation of character type by enumeration (see A5) and a fundamental change in the way subscripts are syntactically thought of in J73 -- they are now simply integers -- and in the way in which the enumeration type is conceived -- they are now simply mnemonic integers. The intent of A6 could be met by including in the table declaration the subscript type (much as PASCAL does) to include integer, status or character.

---

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time. ....P

Alternative structures for records are possible .....P
Discrimination condition may be any Boolean expression .....P

Alternative structures are implemented through specified tables, based variables, or OVERLAY. Discrimination is entirely by convention, there are no language constructs to facilitate discrimination, and arbitrary references to memory are simply achieved. (2.1.5)

The concept of discrimination is so foreign to J73 that to modify the language to include it is essentially to define a new language. The requirement in A7 impacts the three language features (specified tables,

based variables, and OVERLAY) none of which is trivially modifiable to achieve the intent of A7.

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value. ....P

Automatically defined for any type (excert...) .....P
Available for individual components .....I
(Assignment and reference via functions) .....P

Assignment and reference are not consistently defined: Type matching is not imposed for structure assignment or parameter passing and whole arrays are not assignable (only entries of TABLEs, that is, individual instances of a structure). Assignment and reference are defined for simple items (i.e., non-record, non-array types). Assignment to union types is not possible because union type is not defined in J73.

(Reference by function is possible, except that only primitive types are definable as function "results". For example, one cannot define a function which yields a component of a record or of an array or that yields a record or an array. Functions on the left-hand-side are not definable, but certain built-in functions may be used on the left: BIT, BYTE.)

The implementation of consistent assignment and reference is an enormous change to J73, especially to permit array assignment and to add all types as function results (which we construe as essential to the notion of consistent reference). Complete consistency argues for user-definable left-hand-side functions to supplement BIT and BYTE, but mechanizing these is beyond the scope of this study.

---

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity. ....P

Array types may not be compared, and type matching is not imposed for record comparison such that, for example, a table entry containing a character and a real may be found equivalent to an integer. (4.4, 4.5, 4.6, 4.7)

There will be a minimal impact to include array comparisons, but a major impact to include type checking for records (although this is a problem that must, in general, be solved if J73 were to meet the most fundamental intent of Tinman).

---

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration. ....P+

Built-in for all numeric and enumeration types .....T
Ordering can be inhibited when desired .....F

Relational operators <, >, <=, >=, =, <> apply to all numeric types including "status" types. Unordered sets unknown in J73. (4.4, 4.5, 4.6, 4.7).

---

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation. ....P+

Addition .....T
Subtraction .....T
Multiplication .....T
Division with real result .....T
exponentiation .....T
Integer and fixed point division with remainder .....P+
Negation .....T

Primitive operations use familiar notation (+, -, *, /). Division with real result occurs whenever at least one of the divide operands is real; division between integers produces an integer result. A special operator (\) yields the remainder of an integer (or real) division. (4.8)

(Note: Because fixed point is not supported in J73, there is no fixed point division with remainder.)

---

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point

numbers. ....P

Never from the left for data within range .....P
Never on the right for integer and fixed point .....F
Implicit floating point rounding beyond precision allowed .....T
(Run time checks can be avoided) .....N/A

Assuming that "data within range" excludes intermediate results that may be out of range (how else do we get truncation?) then J73 truncates on the right by implementation convention; truncation is not discussed for arithmetic operations except for assignment to floating variables in which one has the option of rounded or truncated (presumably low-order). Right truncation is always implicit for integers.

Since truncation is not properly defined, left truncation could be added at modest implementation cost. Run-time checking could be added to correct the integer truncation problem (but with the option for disabling the checking mechanism) at moderate cost (either by inserting checking code in the object program, or by interpretive execution via debugging data generated as an amendment to the object program).

---

B6. The built-in Boolean operations will include "and", "or", "not", and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode. ....P

Short-circuit and .....P
Short-circuit or .....P
Not .....T
Xor .....T

By convention, most J73 implementations employ short-circuit mode, but not by language definition. (4.5, 4.6, 4.7)

---

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure. ....P

Scalar operations on arrays .....F
Assignment between records and arrays of conformable type .....F-

J73 does not permit assignment or reference to arrays in any context except the passing as an actual "reference" parameter. Assignment between records is permitted with conformability defined between any two records. That is, there is no type checking, and the assignment is by bit string manipulation rather than data equivalence. (Section 4)

There will be a moderate to heavy impact to add array assignment and reference with proper type checking and to add conformability checking for records. (Also discussed under requirements B1 and B2).

---

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors. ....P

No implicit conversions .....I
Explicit between integer, fixed point, and floating point .....T
Explicit between fixed point scale factors .....F
(Explicit between integer and Boolean) .....N/A
(Explicit between integer and enumerated types) .....F
(Explicit between different enumerated types) .....F

Built-in functions (with specification for exception conditions) are provided for INT, FLOAT, CHAR, and BIT of types float, char, signed-integer, unsigned-integer, and bit.

There is no provision for character representation of internal numeric objects nor vice-versa.

Enumeration type in J73 is treated always as integer, thus, no conversion required or defined. (1.7.1.2)

There will be a modest impact to provide the various explicit conversions.

---

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is

truncated. ....F

Implicit conversion between ranges .....F
Exception condition on integer and fixed point truncation .....F

Ranges not defined.

Truncation not noted.

Run-time checks for truncation could be added without conflict with other language features. (See B5.)

---

B10. The base language will provide operations allowing programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent. ....F

Sending and receiving of data .....F
Sending and receiving of control information .....F
Dynamic device assignment .....F
User exception condition control .....F
Installation independence .....F
(Data formatting capability) .....F
(Reading and writing of bit strings) .....F

J73 defines no I/O. (Note: A non-standard superset of level I implemented for Wright-Patterson AFB Avionics Lab includes a FORTRAN I/O like capability).

Modifications to implement I/O could all be made entirely external to the language, i.e., the procedure, function, table, and compool capability well provide all that is required in B10. Note that the sending and receiving of control information is clearly so hardware and operating system dependent that it seems as if this requirement is redundant with A2; that is, the enumeration type is sufficient as an agent for the control information transfer, and language artifacts beyond this primitive capability will suffer in the attempted mapping from real hardware/systems to the language constructs.

---

B11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate. ....F

Union .....N/A
Intersection .....N/A
Difference .....N/A
Complement .....N/A
Membership predicate .....N/A
(Set inclusion) .....N/A

Sets and power sets are not defined in J73.

J73 will accommodate the addition of sets and powersets and the required operations with small impact on existing features and implementations. Some implementation dependent upper bound on set size may be included in a realistic approach (say, w, where w is the word size in bits), and this parameter would be available as an inquiry primitive in the language.

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right. ....F

Side effects must occur in left-to-right order .....F
(Embedded assignments) .....F

The language is equivocal on the subject. J73 specifies that "within a particular precedence level, operations are combined from left to right", while later, "The evaluation order of formulas is unspecified except that address:formulas and indices (subscripts) must be evaluated before variables with which they are associated are referenced". (4.8)

We take the latter as the governing specification.

There is no language impact in specifying left-to-right evaluation order of expressions or formulas (side effects are occasioned, not evaluated); however, existing compilers will require modification to enforce this rule.

---

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized. ....P+

Few precedence levels .....P+
No user-defined precedence levels .....T
Operands of an operation are obvious .....T

J73 defines 10 precedence levels. This is probably more than the "few" required by C2. However, the levels are traditional and familiar. (4.8)

---

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed. ....T

(Section 4)

---

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time. ....I

Note: J73 proscribes constant formulas which contain floating or character operands or which contain the exponentiation operator. (4.9)

---

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to ease of learning. ....F

Parameter rules consistent in all contexts .....I
No special operations applicable only to parameters .....F

In procedure and function calls, constants and simple items may be passed as actual parameters opposite tables (records/arrays) and blocks, which is inconsistent with assignment which is not defined for either tables or blocks. This feature is used to defeat type checking for parameters and implements a kind of "hidden" union as formal parameters and allows the writing of user generic procedures. (2.2.3, 3.10)

There is no impact on the existing language specification to prohibit the passing of such parameters. The impact on existing programs is unknown.

---

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter. ....P

Actual and formal parameters will agree in type .....?
Rank of parameter arrays is fixed at compile time .....I
Parameter array size and subscript range can be passed .....F

See C5 for remarks about formal parameter tables and blocks. Size and subscript range of parameter arrays are fixed by the formal parameter description. (2.2.3, 3.10).

The addition of variable size formal arrays will add moderate cost to the parameter passing mechanism but should have no adverse impact on existing mechanisms for passing fixed arrays.

---

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters. ....P

Act as constants (call by value plus) .....T
Act as variables (call by reference) .....P
Excertion control .....T
Procedure parameters .....T
(Act as variables, but call by value) .....T
(Act as variables, result parameter) .....T

Call by reference parameters are restricted to arrays and blocks. Call by value allows assignment to a formal parameter (which does not alter actual parameter). Exception control is by label parameters. J73 also allows "value output" parameters, but these are restricted to non-composite types, i.e., to scalars, single array components, or single record components. (2.2.3, 3.10)

Reclassifying formal parameters requires a significant change to the syntax of the language; parameters are currently classified implicitly. Doing away with output parameters would require complicating the actual parameter descriptor in order to accommodate passing of record and array components. This would have a moderately heavy impact.

---

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time. ....P+

Above properties optional .....P+
Above properties are fixed at run time .....F

Generic procedures are partially achievable through the use of formal array parameters against which actual parameters of virtually any type may be passed. Thus, the intent of C9 is achieved at the cost of some clumsiness and obfuscation. (2.2.3, 3.10)

This requirement is in such blatant conflict with the strict typing conventions dictated by Tinman, that one is at a loss to suggest a scheme for meeting C9 that is not a hopeless kludge. In fact, the existing J73 mechanism is no worse a kludge than any other (e.g., ALGOL 68 'moce outtype').

---

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time. ....P

Variable number of arguments possible .....F
All but a constant number of arguments have the same type .....F
Number of arguments in each call is fixed at compile time .....T

The number of arguments is fixed for all user-defined and built-in procedures and functions. (2.2.3, 3.10)

The scope of change to permit variable length argument lists is not accommodated within the constructs available in J73. Such implementation approaches as sublists,, for example, require moderate to heavy changes in parameter passing conventions and syntax. The simplest syntactic approach would be to enclose formal sublist parameters in parentheses and to use some variant of subscript notation to select the sublist member, all of which requires a hidden parameter, accessible to the program, that identifies the length of the variable part.

D1. The user will have the ability to associate constant values of any type with identifiers. ....T

The J73 DEFINE permits this, as, for example:

DEFINE DOUBLE'ONE "1.0M62".

(2.5.1)

---

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output). ....P

Literals for all built-in types .....P
Consistent interpretation in program and data .....N/A

J73 provides literals for all non-structured data, but provides no array literals nor record literals. (5.3)

No I/O is defined in J73. (See B10.)

It would be of modest cost to add structured literals to J73. The main problem will be to find some notation that doesn't provoke outrage from some corner or other.

---

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values. ....P

Initial value can be specified as part of the declaration .....P
Initialization occurs at allocation scope entry .....P
No default initial values .....T

The syntax for data declarations includes the facility to specify initial values for so-called "RESERVE" data; data defined as allocated to its containing scope ("IN" data) may not be initialized. For arrays, initial values may be specified by particular subscript; repetition counts are also allowed. Allocation of initialized data occurs at

compile time, i.e., it is globally static regardless of scope of declaration. (2.1.4, 2.1.5, 2.1.7)

Initial values can be trivially provided for "IN" data. It would seem to be decreasing of efficiency to eliminate the global static allocation, although providing dynamic initialization at scope entrance for "RESERVE" comes free if it is done for "IN" data.

---

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types. ....F

Numeric variable range specification mandatory .....N/A
Fixed point variable step size specification mandatory .....N/A
Range and step size specifications do not define a new type .....N/A

J73 does not provide for range specification at all. (Section 2)

There would be moderate cost to add range specification to J73 (e.g., consider borrowing J-3 or Pascal notation). Of greater cost is the addition of range-checking code (See B5, B9).

---

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a continuous subsequence of any enumeration type. ....F

Ranges of an enumeration type are allowed .....N/A
No arbitrary restrictions on the structure of data .....N/A

See D4. (Section 2)

---

D6. The language will provide a pointer mechanism which can

be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables. ....P-

Recursive and network structures provided .....P-
Handles variable-value and structure-component connections .....P-
Pointer property is an attribute of a typed variable .....P
Pointer property not for constants, affects only assignment .....F
Pointer property mandatory for dynamic allocation .....N/A
Allocation scope never wider than access scope .....F
(Either the value or the pointer is modifiable) .....T
(Pointer mechanism handles procedures and parameters) .....P
(Remap and replace assignment have different syntaxes) .....F
(Built-in dynamic variable creation) .....F
(Variable equivalence classes are declarable) .....F

J73 provides for pointed-to things through the based variable -- items, tables, and blocks may be based. References to based variables occur either implicitly through an "address" variable defined in the declaration or explicitly through an "address:formula". The address variable or formula is nothing more than an integer expression that yields a machine address. There is no indelible association between a pointer and the pointed-to object. There is, therefore, virtually no type-checking, and, of course, even machine addressing exceptions are possible with no help from the compiler in detecting these anomalies.

Shared structures (including lists) may be mechanized by declaring record fields which by convention are used as bases or pointers to other instances of the record type. (There are other, perhaps more common and better protected ways of implementing lists in which a field of the record (a table item in J73) is set aside as an index to another instance -- say, the next in the list -- of the record type; all the space is taken from a statically allocated table whose definition is the record prototype.) This is perhaps best seen as what is essentially a machine-language mechanization of list structures (or shared records, networks, etc.) and carries with it the same level of error-propensity and unintended and/or unwise access to values of improper type. J73 does not, therefore, offer any language constructs that make clear and obvious the manipulation of shared or list-structured values. The based mechanism really offers a total escape from type-checking and permits nearly unrestricted access to the most primitive level of the target machine data.

J73 falls short of meeting the D6 requirement in that (a) pointer variables may be totally independent of the pointed-to object, (b) pointers may assume any arbitrary integer value, (c) pointer are not restricted to a property of a typed variable, and (d) the allocation scope can be wider than the access scope. (Section 2)

Because this requirement is so broad and complex, and because the requirement itself is not free of ambiguities, it is not clear how it would impact J73 were its provisions appended to the language. Somewhat superficially addressed, by simply treating the based property as an attribute and eliminating address formulas and restricting the property to tables alone, the intent of D4 could be met at modest cost. The affect of this one existing programs not known.

E1. The user of the language will be able to define new data types and operations within programs. ....P-

Even though the table declaration in J73 is quite flexible, the inability within the language to define records as types or to define records containing arrays severly limits the extensibility desired in E1.

Operations are definable through functions and procedures only. The language does not permit the definition of new infix operators or to redefine existing infix operators.

See the remarks under A2 for data definition enhancements. The addition of infix operator definitions is a minor effort if the type of operands is fixed in the definition; the effort becomes greater if operand mode is used to select the proper operator. In the former case, the existing function mechanism within the compiler serves to implement the infix operator feature; in the latter case, operator selection becomes somewhat more complex, although the Tinman proscription against implicit type conversion eases the selection problem.

---

E2. The "use" of defined types will be indistinguishable from built-in types. ....F

Since defined types are not supported in J73, this requirement is not met.

Once defined types are allowed at all, consistent syntax is almost free. The real effort is in providing an extension to allow prototypes, unions, records of arrays, and the pointer property which are the essential elements missing from J73 to allow true type definition. (See A2.)

---

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations. ....P

Some attributes of type definition may be defaulted in J73: integer size, real precision, table lower bounds, character string size, parameter label definition, allocation class of variables. (Section 2)

It would be a minor change to require complete definition.

---

E4. The user will be able, within the source language, to extend existing operators to new data types. ....F

J73 does not accommodate defined types as required by Tinman.

---

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented. ....F

Construction .....F
Selection .....F
Predicates .....F
Type conversions .....F
Operations and data can be defined together .....F

The requirement of E5 is entirely foreign to J73. In a sense, of course, any user-defined procedure may define a type and the operations for that type; but carrying this notion to its conclusion, we feel, violates the intent of Tinman and would render the language comparison useless.

---

F6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time. ....P-

Enumeration .....P
Cartesian products (records) .....P
Discriminated union .....F
Powerset of an enumeration type .....F

Status type is the only enumeration mechanism but does not define a new type in J73; it is treated instead as (unsigned) integer.

Records are definable by the table mechanism, but also do not define types, nor may they be used as type constructors. (Section 2)

---

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired. ....P-

J73 does not define subsetting, but does allow free unions through specified tables, overlay, and based variables. (2.1.5.7, 1.5, 2.1.4, 2.1.5, 2.1.6)

---

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type. ....F

Initialization .....F
Finalization .....F
Allocation actions .....F
Deallocation actions .....F

See the remarks under E5.

F1. The language will allow the user to distinguish between scope of allocation and scope of access. ....P

Allocation scope is never narrower then access scope. "RESERVE" variables are in effect globally allocated, but access is strictly controlled through lexical scope rules. Based variables, however, may allow an exception in permitting outer scope access to inner scope data structures. (Section 2)

---

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components. ....F

Allowable operations can be limited .....F
Access can be limited where used .....F
External declarations need not all have the same scope .....F
Naming conflicts can be avoided (renaming) .....F

Data types are not definable in J73.

---

F3. The scope of identifiers will be wholly determined at compile time. ....T

The scope rules of J73 conform to the F3 description (1.3.3, 2.3)

---

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language. ....P

Comment: it is hard to understand F4 as a language requirement. The requirement should be that data and operations should be definable by means of a library mechanism. No language can meet the abstract requirement of supporting unspecified applications in its libraries.

(J73/I for WPAFB/AFAL supports a FORTRAN-like I/O capability.)

---

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages. ....P+

Program component libraries accessible at compile time .....I
Libraries can contain foreign language routines .....P
Interface requirements checkable at compile time .....I

Comment: The provision in F5 to include "foreign" routines in the library is clearly not a language requirement if they are obliged to exist in an object code form that is indistinguishable from the object code of the "Common HOL", i.e., in what sense, then, is it foreign? The requirement appears to be directed at the foreign language, not to the common HOL. There is no way sensibly to evaluate existing languages against this part of the requirement.

J73 for WPAFB/AFAL accommodates library routines written in FORTRAN, in which case the J73 object code is generated to conform to FORTRAN convertions. This is an implementation dependent aspect of J73, although directives are provided to guide the compiler in generating special-form linkage to foreign languages.

---

F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program. ....P

Libraries and compools will be indistinguishable .....F
Immediately accessible sublibraries at any level .....P

The accessibility of sublibraries is more a function of the operating system's data management characteristics than of the HOL's library/compool properties.

J73 compools are preprocessed data, define, and subprogram definitions. Libraries are not defined by J73, but in practice, their characteristics are taken from the target machine operating system conventions. (1.6, 2.6.4)

---

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware. ....P-

J73 provides access to certain machine-dependent characteristics by means of inquiry primitives (See I3). However, no mechanism is provided to peripheral equipment as special hardware. (1.2.2, 4.10.10.1)

This requirement is too abstract to provide meaningful modification estimates.

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling. ....P

Sequential execution .....T
Conditional execution .....T
Iteration .....T
Recursion .....T
(Pseudo) parallel processing .....F
Exception handling .....F
Asynchronous interrupt handling .....F
Control structures from a small set of simple primitives .....F

J73 provides GOTO, IF, and FOR statements. Recursion is attainable using features not intended to support recursion ("Based" procedures and the DEF facility). (3.2, 3.4)

GOTO is allowed out of scope, and parameter labels are permitted as J73's solution to exception exits from procedures. This conflicts with G2.

Parallel processing is major impact. Exception and asynchronous interrupt handling could be implemented through the library somewhat trivially if certain constraints were imposed (such as proscriptions against global exits (GOTO) from exception routines).

---

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition. ....F

J73 allows direct branches out of scope and parameter label GOTOs. (3.2, 3.4)

Branches out of scope could be trivially proscribed. Eliminating parameter labels leaves a small gap in providing procedure exception exits.

---

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on

the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives. ....P

Based on Boolean expression .....T
Based on type from discriminated union .....F
Based on computed choice among labeled alternatives .....T
All alternative must be accounted for .....F
Simple mechanisms will be supplied for common cases .....F

J73 does not provide discriminated unions. J73 switch supplies the familiar CASE facility. Unaccounted for alternatives are undefined in J73 (in the SWITCH statement ELSE may be omitted in the familiar way).

There are no simple mechanisms for common cases. (3.7, 3.8)

Implementing discriminated unions must precede implementing computed alternatives based on union type.

---

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted). ....P

Termination can occur anywhere in the loop .....P
Multiple terminating predicates are possible .....P
Entry permitted only at the loop head .....P+
Simple cases are clear and efficient .....T
Control variable is local to the loop .....F
Control value is efficiently available after termination .....F

Termination within a loop is possible by GOTO, RETURN, STOP, or by turning the "WHILE" condition false. (3.9.2)

Multiple terminating predicates are only possible by normal conditional branches within the loop that force exit.

The language permits multiple entry loops by not diagnosing them. However, essentially, J73 calls a branch into a loop undefined (3.9.2).

Control variables may be any non-composite, scalar variable.

The value of the control variable after normal termination is available only as an implementation quirk. Since ordinary variables are used as loop control variables (3.9.1), it follows that on abnormal loop termination, the variable holds its last assigned value. At normal termination, the last assigned value is not specified, i.e., the user cannot know whether incrementation precedes as follows the test. (3.9)

Restructuring loops is a moderate task and quite localized. Compilers should be obliged to diagnose branches into FOR-loops.

---

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure. ....P+

No recursive procedures within recursive procedures .....T
(Maximum depth of recursion can be specified) .....F
(Recursiveness must be specified) .....F

Recursive procedures are defined by based procedures, DEF PROC, and REF PROC. Since full J73 defines recursive procedures, one can see that the recursive capability in Level 1 is unintended. (2.2)

---

G6. The source language will provide a parallel processing capablity. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution. ....F

Able to create and terminate parallel processes .....F
Process can gain exclusive use of resources .....F
No parallel routines within recursive routines .....F
No routines within parallel routines .....F
Maximum number of simultaneous instances are declarable .....F
(Access rules are enforced) .....F

J73 makes no mention of parallel processing.

It would be a very heavy impact to add parallel processing. This requirement is vague and potentially defines an operating system, so to evaluate the scale of the effort to include it in any language at this level of investigation is impossible.

---

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program. ....P-

Program can get control for any exception .....F
Parameters can be passed .....F
Can get out of any level of a nest of control .....F
Can handle the exception at any level of control .....F

J73 does not address exception handling. In practice, there is nothing in J73 that prevents exception handling. The stylized system interface is handled by machine language but J73-callable procedures. While J73 provides no exception handling artifices, implementations written in J73 do permit rational exception handling.

It would be a moderately heavy impact to add exception control constructs and discipline to J73.

---

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation. ....F

Priority specification .....F
Synchronization via wait/enable operations .....F
Wait for end of real time interval .....F
Wait for end of simulated time interval .....F
Wait for hardware interrupt .....F
(Can enable and disable interrupts) .....F

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous. ....P+

Free format with statement terminator .....T
Mnemonic identifiers possible .....T
Based on conventional forms .....T
Simple grammar .....P+
No special case notations .....P+
No abbreviations of identifiers or keywords .....P+
Unambiguous grammar .....T

Special case notations: status constants (2.4.1). Keywords abbreviated: PROC, type specifiers (e.g., F for FLOATING, S for INTEGER, etc.). (2.3.1, 2.1)

---

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences. ....T

J73 does not allow modifications of built-in operations in any way: precedence, operator meaning, and language words are irrevocably fixed by the J73 definition.

---

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset. ....T

The special marks, digits, and letters defined for J73 (5.1) are a proper ASCII subset.

---

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals. ....P+

Break character exists .....I
(Literals are self-identifying as to type) .....P+
(Bit-string literals for any type) .....I

J73 uses the apostrophe for the break character. The only literal that is not self-defining is the unqualified status constant. (1.3.1)

---

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings. ....F

Lines are undefined in J73; the input is a continuous stream of characters.

It would be a trivial change to the language to satisfy this requirement.

---

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used. ....I

J73 defines 68 keywords which are reserved (5.2.1). 15 of these keywords are unused in the Level I subset, leaving a net of 53. This seems few enough.

---

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not

prohibit automatic reformatting of programs. ....P+

Uniform comment convention .....I
Look different from code .....P
Bracketed by one or two characters .....I
Can contain any characters .....I
Can appear anywhere reasonable .....I
Terminated by the end of the line .....F
Compatible with automatic reformatting .....I

Since "any characters" may appear in a comment, comments can indeed look quite a bit like code. (This requirement seems inconsistent).

Comments are delimited by the double quote mark which is also used to delimit define Strings (5.4, 3.5.1).

---

H8. The language will not permit unmatched parentheses of any kind. ....T

---

H9. There will be a uniform referent notation. ....F

Function calls use round brackets, subscripts use square brackets.

The replacement of square brackets in all contexts with round brackets introduces a syntactic ambiguity between replication counts for preset table data and the subscript notation for the table entry to be preset. For example, currently

```
TABLE AA[1:10]F = (AA*2)(0);
```

parses such that (AA*2) is a "number:formula" which represents a replication count (AA must be a named constant, and the count is a compile-time constant expression). Then, the program

```
TABLE AA[1:10]F = [AA*2] (0);
```

identifies the single entry at AA*2 which is to be preset with the initialized value zero. The former case presets AA*2 entries with zero beginning, by default, with the first entry). By this example, the replacement of square with round brackets demonstrates the ambiguous syntax. A new syntax for initialized data would solve this problem at modest cost.

---

H10. No language defined symbols appearing in the same context will have essentially different meanings. ....P

J73 uses the = for both the assignment and equality operators. In addition, the @ is used for both based variable reference and for reentrable procedure calls (which some might argue is consistent -- a so-called based procedure is one whose data space is supplied by the caller, i.e., the procedure data is based but not the procedure itself, which is the reason for judging it inconsistent here). In addition, subscript brackets are used for a variety of purposes whose syntactic context determines the meaning. The same is true for for unqualified status constants. Lastly, parentheses are used to delimit constant expressions, without which there would be introduced syntactic ambiguity in initialized data constructions.

The language could made more consistent at probably modest cost, but many would object loudly to the elimination of unqualified status constants, and the subscript cases merit more examination than is affordable in this study. Clearly a new assignment operator is a trivial change, and the based procedure call question is arguable in any event.

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program. ....P

J73 defaults the range and precision of numeric data. However, the language does provide inquiry primitives. (2.1.3.2, 1.2.2)

Similarly, the internal representation, character size, and collating sequence for the character type are implementation dependent defaults (which the programmer may not override).

Default range and precision is trivially eliminated. The character type internal representation, collating sequence, and size defaults present a more difficult problem.

---

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary. ....P

Defaults specified for don't care cases .....F
Programmer can override the defaults .....P

The programmer may override size and precision defaults, but not those relating to the character type. (See I1.)

---

I3. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration. ....P+

J73 provides a variety of target machine parameters as language primitives. These include (1.2.2)

Bits per character
Bits per word
Address units per word

Number of characters per (packed) word
Bit size for address formulas
Lowest and highest addresses of statically
allocated data space

In addition, the language provides inquiry functions for (4.10.10.1) size of allocated data for "based" procedures and the bit size, byte size, and word size of data structures (items, tables, blocks).

---

I4. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled. ....P-

Conditional compilation is possible only through the use of special directives (!SKIP, !BEGIN, !END). The conditionality is not expressed through CASE or IF-like constructs.

Moderate to heavy cost to implement compile-time variables and predicates, depending of course, on the sophistication of the control constructs (are FORs allowed, for example?)

---

I5. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language. ....I

J73 Level I is the basic subsets. (This requirement sounds like the basis for extensibility, in which sense, J73 partially fails except for that provided through procedures/libraries/compools.)

J73 Level I offers few redundancies (IF and SWITCH, for example) and these are useful.

---

I6. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition. ....F

J73 leaves compiler capacities/limitations as almost totally implementation dependent.

Setting compiler limits within the language definition may have significant impact on existing implementations which are unknown at this writting.

---

I7. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator. ....T

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs. ....P

No efficiency cost for unused features .....P
Efficient code can be produced for all features .....P

The language implies certain costs for little-used or unused features (procedure calls to handle based procedures and parameter procedures), although the language does not exclude clever implementations that may circumvent these apparent costs.

---

J2. Any optimizations performed by the translator will not change the effect of the program. ....I

Insofar as it is provable that successive translations of the source to object environment yield an equivalent program, then to that degree optimization will not change the effect.

---

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions. ....P-

J73 does not support machine code insertions, but there is nothing to prevent procedure calls to programs written in machine language.

---

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator. ....P+

Encapsulated specification of representation possible .....I
Space/time tradeoff can be specified .....P

J73 permits the programmer to allocate the components of a structure down to the bit level through the use of specified tables (2.1.5.7). Allocation can be controlled to machine address level with the OVERLAY statement (1.5.1).

Space-time tradeoffs are specified through table/table-item packing specifiers to yield unpacked/fast-access, somewhat-packed/fast-access, or tightly-packed/slow-access alternatives (2.1.5.5.2).

---

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics. ....P+

Open/closed properties can be specified .....P+
Open and closed versions have the same semantics .....P

J73 defines only closed routines as procedures, functions. However, the parameterized define mechanisms (2.5.1) may be used to obtain the open routine effect.

## Extraneous Features

We recommend that the following features of JOVIAL J73 (Level I), not required by the Tinman, be kept:

* Character string type (assuming that the Tinman requires only a character type, not a character string type).
* Result parameters (output parameters) for procedures.
* Control of the internal structure of records through (1) specifying the internal arrangement of the record as either parallel or serial and (2) specifying the compiler packing algorithm to be used.
* The REDUCIBLE directive, which allows optimization of function calls which cannot be done without the directive.
* The functions ABS and SIGN.

The following features provide a useful capability. They should either be kept intact or replaced by some equivalent feature:

* Bit string type. The bit string constant appears to be overly elaborate, however.
* The SHIFT function, used to shift bit expressions.
* The bit-manipulating operators NOT, AND, OR, XOR, and EQV.
* The trace directive. This debugging aid could well be expanded.

The following features are machine dependent, but useful. They should be retained because of this usefulness, but it should be required that any use be bracketed (encapsulated) by statements which mark them as being machine dependent (requirement J3):

* The functions BIT, BITOF, BYTE, and CHAR, which permit accessing bits and bytes of expressions.

The following features of JOVIAL J73 (Level I), not required by the Tinman, should be deleted:

* The various size function (BITSIZE, BYTESIZE, WORDSIZE), which return the number of storage units

allocated to various entities. This is too hardware dependent.

* Based data and procedures and the related features of the language (e.g., the DSIZE function and the interference directive). This is an area of programming entirely foreign to the Tinman and, until the Tinman addresses the subject, these features should be removed.

* The function LOC, which returns the machine address of an entity. This is too hardware dependent and its primary function is probably that of a pointer.

* OVERLAY, a form of free union.

* The ability to return through more than one level of procedure linkage by executing a single RETURN statement. While perhaps useful in some cases, this tends to obscure program flow and thereby drive up maintenance costs.

* Blocking of data. This is properly a function of a loader or link editor.

## Summary

As a result of evaluating J73 against the Tinman requirements, several good features of the language which satisfy or come close to satisfying these requirements have been identified. The language requires type checking of all entities at compile time, permits precision specification for individual variables, requires that the number of dimensions, type, and lower subscript bound of arrays be fixed at compile time, permits relational operators for comparing all primitive data types including enumeration types, allows various built-in arithmetic operations including integer division with remainder, permits explicit conversion between various data types through functions such as INT, FLOAT, CHAR, and BIT, and establishes a few, easy to understand rules for expression evaluation and operator precedence. Rules for handling parameters for procedures and built-in operators are consistent throughout the language definition. The language provides for exception control parameters (i.e., labels) as well as procedures as parameters. It has the capability of assigning identifiers to constants. It permits initialization of most data types at the time of their definition. It allows the definition of recursive procedures and recursive data structures with the use of a pointer mechanism. The language also requires that all items, arrays, tables, etc., be explicitly declared before use, hence that no implicit declarations will be permitted. The scope of these identifiers is also determinable at compile time. The user can have access to both compools and libraries. The syntax of the language is consistent, allows free formatting, and does not provide for special or rare cases. The language syntax cannot be modified by the user or the implementor in any way. Single quotation is permitted as a break character for use within names, thus allowing for mnemonic names to be used as identifiers and to increase readability of programs. The language has 68 keywords, all reserved. J73 assigns a unique meaning to most symbols; the only exceptions are =, which is used to denote both assignment and logical identity, and @, which is used for both based variable reference and for reentrant procedure calls. No implementation dependent defaults are permitted with the exception of range, precision of numeric data, the internal representation of character type, and the collating sequence. The programmer has the facility to override size and precision related defaults. J73 provides a variety of target machine related language primitives. The user can specify the bits per character, bits per word, number of characters per packed word, lowest and highest addresses of static data storage, etc. The language is designed around a level 1 kernel and provides for extensibility. It also permits user defined open and closed routines.

There are, however, many characteristics in which J73 falls short of the Tinman requirements. Primary among these are the lack of parallel processing, exception handling, and input/output capabilities. There is no facility in the language which can provide global specification of precision for arithmetic, range of variables, enumeration of unordered sets, and truncation of numbers from the right instead of the left. The language does not allow definition of

procedures with a variable number of arguments, nor does it define scalar operations for arrays. J73 does not allow the user to define new data types or new operators. Compools and libraries are treated differently. The GOTO statement is unrestricted in its use and allows transfers of control into and out of the scope in which it appears. The loop control variable is not local to the loop and has to be defined before use; that is, outside the range of the loop. The concept of uniform referent notation is ignored by the language; the syntax of function calls is different from array subscript notation. Moreover, the functions or procedure names cannot be used on the left hand side of the assignment statement like other data variables. The language definition also fails to specify machine independent compiler limitations.

Language designers have two alternatives in modifying J73 to meet the Tinman requirements. The first is to alter, delete, or extend the language capabilities, use the existing syntax and semantics as the base, and arrive at the desired language. The second is to draw upon the strengths of J73 conceptually, and to design a new language from scratch.

The first alternative will require the addition of many features, such as tasking capabilities, exception handling by means of control features or other suitable mechanism, a complete input/output capability, ability to define new data types and operators, including the extension of new operators to built-in data types. Extensive modification of the pointer capability is required, because of the rudimentary nature of the current mechanism. In particular, options should be provided to protect the areas where pointers are pointing during dynamic allocation. Other modifications such as provision for range option in data definition, truncation of data from the right instead of the left, dynamic as well as static allocation of arrays, variable number of arguments in procedure definitions, etc., will have to be provided for. Certain other features of the language, such as automatic default substitutions for integer size, real precision, table lower bounds, etc., will have to be discarded. The overall modification process for J73 will be of major proportions, but it can be accomplished. The syntax of the language obtained by this process will certainly be clumsy at places since it contains a blending of the old with the new.

Development and definition of a new language to meet the Tinman requirements will be a major task indeed, but it has a better chance of arriving at a language which is consistent and elegant in its syntax, is modular and extensible, as is capable of efficient implementation. Many of the good and useful concepts of J73 can be included in the design of such a language.

Regardless of the alternative used, the new language will be significantly different from the existing J73 and will probably not allow existing J73 programs to be easily modified to run under the new version of the language. In such a case, there seems to be no point in claiming any relationship between the new language and J73.

# A COMPARISON OF
# PEARL
# to
# TINMAN

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

## Introduction

This report gives a comparison of the language PEARL to the Tinman language requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1976, Section IV). For the purposes of this comparison, PEARL is considered to be defined by:

PEARL
Subset for Avionic Applications
Language Description
June 1976
FSG Elektronik-System-Gesellschaft mbH

Tinman contains 78 language requirements. This report compares PEARL to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used. (Occasionally no text is needed if a requirement is totally satisfied.) If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - Only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

The symbols P, P+, and P- will often be used with requirements which

are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

The report concludes with two summaries. The first is of the features of PEARL which are extraneous to Tinman and the desirability of retaining each of them. The second is of the language as a whole and the desirability of modifying it to bring in into line with the Tinman requirements.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs. ....T

PEARL requires that all variable data must be defined by a data name and a data type. Constant data that are to be associated with identifiers must be similarly declared and must also contain the invariable (INV) attribute. The language also requires the types of arrays and each element of a data structure to be specified by the user. Hence the language meets this requirement. (pp. 38-47)

No conflict with other requirements.

---

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators. ....P

Integer .....T
Floating Point .....T
Fixed Point .....F
Boolean .....P
Character String .....T
Arrays .....T
Records .....P

The PEARL language definition permits integer data type (called FIXED), and provides for floating point, character string, array, and structure (record) type, although structures cannot contain array components. (Arrays of structures are permitted.) There is no fixed point type. Most of the functions of Boolean are satisfied by bit type of length one, but there are no Boolean constants.

Addition of the missing types would require a modest effort in most compilers. For fixed point the hardest problem is the definition of acceptable scaling rules. The addition of structures containing arrays and a Boolean type is relatively easy.

---

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code. ....P

Global arithmetic precision specification mandatory .....U
Individual variable precision specification permitted .....P

Floating point variables in PEARL can be given a length attribute (LENGTH), which specifies the number of bits in the mantissa of the internal floating point representation. This gives something of an ability to specify precision, but it falls short because there is no acknowledgement of the different normalizations extant (binary, octal, hexadecimal, etc.). One assumes that this specification is the minimum number of bits in the mantissa, but the manual has no explicit statement to this effect. These specifications can be made globally and individually, but the manual does not say that either is required.

It would be a simple modification to the language to require floating point precision specifications, and only a simple modification to compilers if they are to be interpreted as a directive to the compiler when it chooses between various precisions at its disposal on the target machine. If, however, they are meant to have some impact on the encoding of comparisons and expression evaluation -- as suggested by the Tinman elsewhere -- then the cost becomes significantly higher.

---

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler. ....F

Treated as exact quantities .....F
Range and step size determined at compile time .....F
Scaling handled automatically .....F

Fixed point, in the Tinman sense, is not supported at all by PEARL.

The addition of fixed point requires a moderate effort in most compilers. It is doubtful that treating fixed point data as exact quantities is practical, however.

---

A5. Character sets will be treated as any other enumeration type. ....F

New sets can be defined as enumeration types .....F
ASCII and EBCDIC are provided .....F
(Conversion capability between sets is available) .....F

The language does not provide the user the capability to enumerate new character sets, hence it does not provide any conversion capability between sets. Conversion routines between ASCII and EBCDIC are not required in the language definition, and if present in the library, are an accident of implementation. The language defines its character set to consist of 26 alphabetic characters (A-Z), 10 digits (0 through 9) and 15 special characters. Furthermore, it allows installation dependent special characters to be added to the character set. (pp. 31)

To met this requirement the language must provide for some facility (e.g., Status Variables) which can allow a user to define his own character set for a given program. The library must support the conversion routines to convert the user specified character-set into the natural set for the machine on which the compiler is implemented. This should include routines which can convert ASCII code to EBCDIC and vice versa.

Implementation of this facility will affect all existing compilers. An appropriate place in the program will have to be established to define new character sets. All processing for character strings, literals, etc., shall have to be modified to provide for the new character set. Several conversion routines will have to be added to the library.

---

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope. ....P

Number of dimensions is fixed at compile time .....T
Type is fixed at compile time .....T
Lower subscript bound is fixed at compile time .....T
Upper subscript bound is fixed at scope entry .....F
Subscripts only integers or from an enumeration type .....P
Subscripts will be from a contiguous range .....T

The user is required to specify the number of dimensions of the array at compile time as well as the type and the upper bound of the subscript. The lower subscript bound is fixed at 1. The user-provided upper subscript bound must be an integer constant. Subscripts are integers only.

To fully meet this requirement the capabilities of fixing upper subscript bounds at scope entry rather that compile time and using elements of an enumeration type as subscripts must be added to the language. Both of these would have only a moderate cost, with the cost of the latter being a part of the cost of adding enumeration types in general.

---

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time. ....F

Alternative structures for records are possible .....F
Discrimination condition may be any Boolean expression .....F

Alternative structures are not supported by PEARL at all.

The addition of such a capability typically has a heavy impact on the syntax of any language, but beyond this the costs (e.g., to the storage allocator and code generator) are only moderate.

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value. ....P

Automatically defined for any type (except...) .....P
Available for individual components .....T
(Assignment and reference via functions) .....F

The language defines the assignment and reference operations for numeric, bit, character, duration, clock, etc., types of data. These operations are also defined for elements of arrays and structures, but not for conformable arrays or structures. Functions can only be referenced, not assigned. (p. 71-72)

To extend assignment to conformable arrays and structures would require a minimal effort. Allowing assignment to functions would require substantially more work.

---

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity. ....P+

The equality operator, ==, is provided in the language for logical comparison of numeric, character, bit, or time type of data. Comparisons of arrays or records (structures) is not defined. A comparison results in a bit (not necessarily realized) being set or cleared: the value is set to '0'B if the relationship is false (not equal). (p. 66)

The addition of array and record comparison requires a minimal effort.

---

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration. ....P

Built-in for all numeric and enumeration types .....P
Ordering can be inhibited when desired .....F

PEARL provides for EQ, NE, LT, GT, LE and GE relational operators for comparison of numeric data types. However, enumerated data types are not supported at all. (p. 66-67)

To meet this requirement the language must provide for status variables and extend the definition of all relational operators to the enumerated data types. Mechanisms should be provided in the language to inhibit relational operations wherever unordered sets are intended.

Implementation of these modifications will affect all existing compilers of the language. The syntax analysis, expression processing, and logical expression processing portions of the compiler will be affected to a moderate extent.

---

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation. ....P+

Addition .....T
Subtraction .....T
Multiplication .....T
Division with real result .....T
Exponentiation .....T
Integer and fixed point division with remainder .....F
Negation .....T

All the operations specified in this requirement are included in the language with the exception of integer division with remainder. (pp. 60-61)

A new operator for integer division with remainder needs to be defined and included in the language. Minimal changes will be required in the language syntax and expression processing routines to implement this feature.

---

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point

numbers. ....U

Never from the left for data within range .....U
Never on the right for integer and fixed point .....U
Implicit floating point rounding beyond precision allowed .....U
(Run time checks can be avoided) .....U

The language does not specify rules for truncation. It is possible to declare the length, in bits, of integer quantities, so it appears that truncation on the left is possible. Since no language defined constructs are provided for range or precision specification, no rules are specified for rounding beyond user specified precision. All details of truncation and rounding are implementation dependent.

To meet this requirement the language will first have to provide for RANGE and PRECISION specification. Then language will have to specify rules that truncation should take place only from the right. For intermediate computations, rounding for floating point numbers should take place beyond the precision specified by the user.

Implementation of these features in the language syntax and expression processing routines will require moderate changes.

---

B6. The built-in Boolean operations will include "and", "or", "not", and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode. ....P

Short-circuit and .....P
Short-circuit or .....P
Not .....T
Xor .....F

The language provides for AND, OR and NOT Boolean operators but does not provide for XOR. The language also does not explicitly specify that the evaluation of expressions containing AND and OR should be done in short circuit mode, nor does it forbid it. Hence the evaluation of these operations is implementation dependent. (p. 65)

The Boolean operations should include an XOR operator. The definition of the AND and OR operations should explicitly emphasize that short circuit mode is required. Some of the existing compilers which are not currently utilizing short circuit evaluation will have to be modified slightly.

---

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure. ....F

Scalar operations on arrays .....F
Assignment between records and arrays of conformable type .....F

The PEARL language definition does not allow arrays and structures to be referenced as a whole (pp. 42,45) except as procedure parameters and at initialization. This implies that scalar operations (e.g., +, -, *, /) cannot be performed on arrays as a whole and must be restricted to their individual elements. A similar interpretation forbids assignment operation to be performed between conformable arrays or records as a whole.

To meet this requirement the language must extend the definition of scalar operations and assignment to conformable arrays and structures as a whole. This may require restriction of numeric operators to only the numeric fields. It may also require special definitions for * and / operators for multidimensional arrays (e.g., the * operation for two-dimensional arrays should be different from matrix multiplication).

Implementation of extended definitions of these operators will require minimal effort in the existing compilers.

---

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors. ....P

No implicit conversions .....F
Explicit between integer, fixed point, and floating point .....T
Explicit between fixed point scale factors .....F
(Explicit between integer and Boolean) .....T
(Explicit between integer and enumerated types) .....F
(Explicit between different enumerated types) .....F

The language allows implicit conversions during assignment operations if the mode of the expression on the right is different from the mode of the symbol on the left. These conversions include automatic conversions from floating to fixed (integer), floating to bit, and

adjustments for variables of different bit and character lenoths (pp. 71-72). Operators for explicit conversions (e.g., FLOAT, FIX, CHAR, BIT, FIT) are also provided (p. 69). Fixed point is not supported at all.

The language definition should forbid all implicit conversions, including the ones for assignment statement. Built-in operators should be provided in the language for conversion between fixed point scale factors at the time a fixed-point capability is added. These modifications to the language definition and the existing compilers will require a minimal amount of effort.

---

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated. ....F

Implicit conversion between ranges .....N/A
Exception condition on integer and fixed point truncation .....F

Since the language does not permit user specification of ranges, the question of conversion between ranges does not arise. The language also does not require that an exception condition be raised when an integer or fixed point quantity is truncated during computations. These aspects are implementation dependent by the language definition.

To meet this requirement the language definition has to permit user specification of ranges. Then it should allow automatic conversion between these numerical ranges. The language definition should also require a run time exception condition to be raised if an integer or fixed point value is truncated.

Minimal changes to the language syntax and existing implementations will be required to implement these modifications. The code generator, the syntax checker and the expression processing routine will be affected.

---

B10. The base language will provide operations allowing programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not

be installation dependent. ....T

Sending and receiving of data .....T
Sending and receiving of control information .....T
Dynamic device assignment .....T
User exception condition control .....T
Installation independence .....T
(Data formatting capability) .....T
(Reading and writing of bit strings) .....T

The PEARL language provides excellent I/O facilities. These facilities have to be specified by the user in the system division and the problem division. In the system division the user specifies the standard peripherals, the process peripherals, the sensors, effectors, interrupts, signals, and the device connections. Accessing and transmitting of data, opening and closing of files, using INTERRUPT, END-OF-FILE, and SIGNAL-IDENTIFIER handling, etc., are done in the problem division. These two divisions combined allow the user to open or close files (using OPEN and CLOSE), declare devices (e.g., DCL (SWITCH, LAMP) DEVICE SINK BIT(1) GLOBAL;) and interrupts (e.g., DCLIR INTERRUPT GLOBAL;), connect I/O channels to devices, create, delete, access, and protect files (using SIGNAL-IDENTIFIERS), send or receive data directly from devices (e.g., TAKE, SEND etc), format and deformat data, and monitor graphic I/O. The user has the ability to connect any channel to any device, thus making his program largely installation independent. Only portions of the system division need to be changed when a computer or installation is changed. (pp. 100-131)

No conflict with other requirements.

---

B11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate. ....F

Union .....F
Intersection .....F
Difference .....F
Complement .....F
Membership predicate .....F
(Set inclusion) .....F

The PEARL language does not support any user-defined data types and hence does not permit the powerset operations listed in this requirement.

Definition of these powerset capabilities is well understood and implementation will require a relatively small effort.

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right. ....F

Side effects must occur in left-to-right order .....F
(Embedded assignments) .....F

The language manual makes no statement about the order of evaluation of operands.

Function references are the only PEARL item capable of having side effects. To impose a left-to-right order of function evaluation is usually a minor problem in any compiler.

---

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized. ....T

Few precedence levels .....T
No user-defined precedence levels .....T
Operands of an operation are obvious .....T

The language establishes precedence levels for binary and unary operators (pp. 60-69) for expression evaluation. It establishes certain well-understood rules for interpretation of expressions. The user does not have the authority to alter or define his own precedence levels for these operators. The language definition clearly defines the operands for each of the operations. Hence the language fully meets this requirement.

No conflict with other requirements.

---

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed. ....T

Wherever, in the language definition, both constants and variables are allowed, expressions are allowed to be substituted in their place.

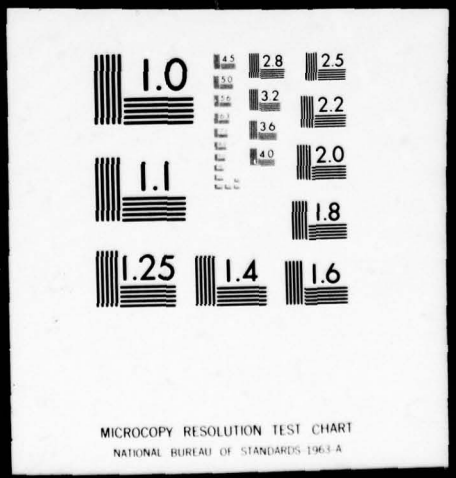
AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

22 of 28
ADA037634

22 OF 28

ADA037634



No conflict with other requirements.

---

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time. ....F

The syntactic definition of PEARL forbids use of constant expressions in place of constants, especially in declarations and specifications (pp. 43, 54, 55, etc.). Only INTEGER-CONSTANT is allowed to be used while declaring a fixed point, floating point, bit, or character type variable; not any type of expression.

The language definition should be made flexible to accommodate constant expressions in place of constants in declarations and specifications.

The necessary changes in the syntax are trivial but it could be moderately expensive to implement in any compilers which do not already have at least a rudimentary form of this capability.

---

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to ease of learning. ....T

Parameter rules consistent in all contexts .....T
No special operations applicable only to parameters .....T

PEARL satisfies this requirement fully. However, only procedures use parameters in PEARL, so the requirement is almost satisfied vacuously. The properties of parameters for subroutines (call procedures) and functions (function procedures) are identical. (pp. 91-97)

---

C6. Formal and actual parameters will always agree in type.

The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter. ....P+

Actual and formal parameters will agree in type .....T
Rank of parameter arrays is fixed at compile time .....T
Parameter array size and subscript range can be passed .....F

The actual and formal parameters must agree in number and type. If the actual parameter is an array, the corresponding formal array must have the same number of subscripts and elements. Hence dynamic arrays are ruled out in the language definition as parameters and the size and subscript range cannot be passed as a parameter for dynamic arrays (p. 92).

To meet the Tinman requirements the language must provide dynamic arrays. This will permit passing array size and subscript ranges as parameters in procedure calls. Implementation of this feature will require a moderate amount of effort and will affect the parameter checking mechanisms, code generation, run-time storage allocation for arrays, etc., in existing compilers.

---

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters. ....P

Act as constants (call by value plus) .....T
Act as variables (call by reference) .....T
Exception control .....F
Procedure parameters .....F
(Act as variables, but call by value) .....T
(Act as variables, result parameter) .....F

Transfer of information between actual and formal parameters can be performed in two ways: (1) call by value and (2) call by reference. In the first case the INV attribute may be specified for the formal parameter, which causes it to behave as a constant on each reference. Exception handling parameters (e.g., labels) are not permitted in the language definition, nor is a user allowed to pass a procedure name as a parameter. Hence the language only partially meets this requirement. (pp. 93-94)

The language definition will have to be extended to permit exception control parameters (e.g., labels) to be included in the actual and formal parameters. It should permit other procedures to be included as parameters as well. Rules for procedures to be passed as parameters will have to be established (e.g., recursive calls, levels of recursion, entry points, etc.).

---

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time. ....F

Above properties optional .....F
Above properties are fixed at run time .....F

Of the listed properties, PEARL parameters have only type, dimension, and format. All such properties are fixed at compile time.

The language definition will have to be modified to allow for the range, precision, and scale specification of variables and parameters at the user's option. The ability to include the array dimension and format in the formal parameter should also be included. All of these properties must be determinable at run-time.

Implementation of these modifications to the language definition will require moderate effort to change the existing compilers. These changes will affect the syntax analyzers and the code generators.

---

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time. ....F

Variable number of arguments possible .....F
All but a constant number of arguments have the same type .....F
Number of arguments in each call is fixed at compile time .....T

The language does not permit user-defined procedures to have a variable number of arguments.

The addition of such a capability would require a moderate effort, both in modifications to the language itself and to existing compilers.

D1. The user will have the ability to associate constant values of any type with identifiers. ....T

Variables, array names and structure names can be used to represent constant values, and can be protected against modification by the attribute INV (p. 47). An entity containing the INV attribute in its declaration must also have the INIT attribute unless the entity is a formal parameter.

No conflict with other requirements.

---

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output). ....P-

Literals for all built-in types .....T
Consistent interpretation in program and data .....U

The language provides definitions and syntax for integer, floating point, bit string, character string, clock, and duration type literals. The language manual makes no explicit statement about consistency of interpretation between program constants and data. Therefore all the dangers alluded to in this requirement are present. (pp. 32-37, 127)

It is trivial to change the manual to require consistency of interpretation but, as the Tinman infers, this can cause amazingly expensive problems in compilers.

---

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values. ....P+

Initial value can be specified as part of the declaration .....T
Initialization occurs at allocation scope entry .....T
No default initial values .....P-

Variables, arrays and structures can be given initial values when they are declared by using the INIT attribute followed by an expression.

The value of the expression must be of the same mode as the data items being given the initial value. The values of variables occurring in the expression must be known when the initializing declaration is made. This implies that initialization can occur at allocation scope entry time.

In the cases of arrays and structures, if there are more data elements than values, then the surplus elements are initialized, by default, to the value of the last entry in the list (p. 46). No default initialization of other variables is specified in the manual. The manual emphasizes that the user must assign some value (either by assignment or initialization) to variables before using them (p. 41).

The default initialization of array and structure elements in cases where the data elements exceed the initial values in number should be forbidden. The user must be made responsible for initialization or assignment of values to variables before using them.

Trivial changes to the language definition and existing implementations will be required to meet this requirement.

---

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types. ....F

Numeric variable range specification mandatory .....F
Fixed point variable step size specification mandatory .....F
Range and step size specifications do not define a new type .....N/A

The language definition does not require nor permit explicit user specification of ranges for numeric variables, nor does it provide a facility for step size specification for fixed point numbers. The language completely fails to meet this requirement.

To meet this requirement PEARL will have to modify the language definition to allow user specification of the range of variables at the time of declarations. It must also require step size specification for fixed point numbers.

Moderate changes to existing implementations are required to accommodate these language modifications.

---

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a contiguous subsequence of any enumeration type. ....F

Ranges of an enumeration type are allowed .....F
No arbitrary restrictions on the structure of data .....F

PEARL does not meet the first part of the requirement because it does not permit user specification of ranges. It does not meet the second part of the requirement either since it does not allow arrays to be part of structures and vice versa.

To meet this requirement the language must permit the range of values which can be associated with a variable, array, or structure to be any built-in or user defined type. Furthermore, the user should be able to make arrays a part of structures and vice versa.

All existing implementations will have a major impact to incorporate these language modifications. The ranges and the use defined data types have to be included. Then necessary modifications have to be made in the structure declarations, array declarations, syntax analyzer, storage allocation, and code generation portions of the compiler to incorporate these features.

---

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables. ....F

Recursive and network structures provided .....F
Handles variable-value and structure-component connections .....F
Pointer property is an attribute of a typed variable .....F
Pointer property not for constants, affects only assignment .....F
Pointer property mandatory for dynamic allocation .....F
Allocation scope never wider than access scope .....F
(Either the value or the pointer is modifiable) .....F
(Pointer mechanism handles procedures and parameters) .....F
(Remap and replace assignment have different syntaxes) .....F
(Built-in dynamic variable creation) .....F
(Variable equivalence classes are declarable) .....F

PEARL has no pointer mechanism at all. Hence, the language fails to meet this requirement completely.

The language definition will have to be modified to include a pointer mechanism which is capable of being associated with each type of variable, can be modified and permits modification of the variable or structure pointed to, is capable of being assigned to, and which does not permit access from outside the scope of its allocation.

Implementation of these language features will require a moderate to heavy effort on existing compilers in the areas of data types, storage allocation, assignment statement processing, data definition, etc.

E1. The user of the language will be able to define new data types and operations within programs. ....F

No facilities are available in PEARL to allow the user to define new data types or new operations.

Modification to language definition and syntax is required to allow user definition of new data types and operations. These operations can be in the form of procedures, but the language must provide the type checking facilities to these defined operations and data types.

Major modifications to existing compilers will be required encompassing the lexical and syntax analysis routines, the expression processing routines, and the type checking routines, to implement these language features.

---

E2. The "use" of defined types will be indistinguishable from built-in types. ....F

The language does not provide for user-defined data types and hence does not meet this requirement.

The syntax of user defined data type and operations as well as their usage should be treated the same way as built-in data types and operations are treated in the language. This will allow uniformity throughout the language and permit the difference between the built-in types and defined types to grow dim in the eyes of the users.

All existing compilers for the language will be affected to provide for this uniformity of built-in and defined types. These features will require minimal effort assuming they are implemented with those required to implement requirement E1.

---

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations. ....P+

The language requires that all variables, arrays, structures, I/O facilities (e.g., devices, channels, etc.) be declared in the program or procedures before use. For global definition the language also requires these entities to be specified in the procedures in which they are being used after having been declared in the main program.

Although definitions may occur in all areas of a program (i.e., module, task, procedure, and block), certain definitions are only permitted in certain areas (p. 38). For instance, a task, procedure, device, file, semaphore, interrupt, or signal definition can only occur in a module, while a BEGIN block can only be defined in a task, procedure, or another BEGIN block.

There are a small number of pre-defined mathematical functions for which no specification is necessary.

To require that the pre-defined functions be specified before use is a trivial problem, if that is truly wanted.

---

E4. The user will be able, within the source language, to extend existing operators to new data types. ....F

Because definition of new types is impossible, this facility is currently not available in the language.

While defining the facilities of user-defined data types and operations (see requirement E1), the language definition should also permit the use of existing built-in operators with user-defined data.

Assuming that this facility will be implemented in conjunction with the facilities specified in requirement E1, a moderate effort will be required to include it in existing compilers.

---

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented. ....F

Construction .....F
Selection .....F
Predicates .....F
Type conversions .....F
Operations and data can be defined together .....F

The language does not have any facility for defining new data types.

As part of the facility for defining new data types and operations specified in requirement E1, the language definition should also require that the user specify the operations permissible on those data. The defined types should not automatically inherit the operations of their constituent built-in types. The definable operations will include constructors, selectors, predicates and type conversions.

A moderate effort will be required to implement this feature, along with those called for in requirement E1, in the existing compilers.

---

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time. ....F

Enumeration .....F
Cartesian products (records) .....F
Discriminated union .....F
Powerset of an enumeration type .....F

None of these mechanisms can currently be used for defining user defined types.

The language definition should be modified to include data definition through enumeration, Cartesian products, discriminated union, and powersets. Implementation of these mechanisms in the existing compilers will require moderate effort.

---

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired. ....T

PEARL permits no form of free union, including OVERLAY.

---

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type

and the actions to be taken at the time of allocation and deallocation of variables of that type. ....F

Initialization .....F
Finalization .....F
Allocation actions .....F
Deallocation actions .....F

The current definition of the language does not meet this requirement, because it has no facility for defining new data types.

While defining the new data types the language should provide for initialization of these data, and user specification of actions to be taken for allocation, deallocation and at termination.

Moderate effort will be required to implement these facilities at the same time as those called for in requirements E1, E4 and E5.

F1. The language will allow the user to distinguish between scope of allocation and scope of access. ....T

The language defines global, local, and block allocation scopes for variables, arrays and structures. The system division allows the user another means for specifying allocation scopes for process signals and peripherals. The access scope for these variables is either equal to or less than their allocation scope. For example, if a variable is defined both at the global and local level, the local definition prevails and the global variable becomes inaccessible in that scope. Since the language does not allow pointers, it is not possible from an outer scope to access variables which were allocated in an inner scope. Hence the language meets this requirement.

No conflict with other requirements.

---

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components. ....P-

Allowable operations can be limited .....F
Access can be limited where used .....F
External declarations need not all have the same scope .....F
Naming conflicts can be avoided (renaming) .....T

Since the language does not allow user definition of new data types, it does not support the first part of this requirement. However, the renaming capability is implemented in the language through the IDENT attribute. Thus the language only partially meets this requirement.

The language must be modified to support user definition of new data types, and it must restrict the operations permissible on these types so that changes to the defined types do not alter the meaning or effect of the calling program.

This modification to the compiler will require minimal effort when implemented along with the features called for in requirements F4 and E5.

---

F3. The scope of identifiers will be wholly determined at

compile time. ....T

The language permits four levels of scope: System, global, local, and block (pp. 52, 75, 101). These must be specified by the user at compile time. Each identifier will have only one definition in each scope.

No conflict with other requirements.

---

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language. ....T

The language supports the notion of libraries rather uniquely. It permits specification of library names for variables, arrays, procedures, and other entities through the GLOBAL-QUALIFIER option (pp. 52, 88). It also requires that compiler libraries be provided to contain certain standard functions (pp. 92). Other routines related to applications can also be stored using any one of these options.

No conflict with other requirements.

---

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages. ....P

Program component libraries accessible at compile time .....T
Libraries can contain foreign language routines .....F
Interface requirements checkable at compile time .....T

The language allows program libraries to be available at compile time. These libraries are capable of holding various entities that are definable in the language (e.g., data, variables, procedures, and tasks). However, the language manual does not define interface with routines compiled from other languages, including assembly language routines.

The interfaces with procedures written in languages other than PEARL should be defined. This may require processing of header information in the compiled routines.

A minimal effort will be required to incorporate these features in existing compilers which check the interfaces already, but to add such a capability to an existing compiler could be quite expensive.

---

F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program. ....I

Libraries and compools will be indistinguishable .....I
Immediately accessible sublibraries at any level .....I

Since the GLOBAL-QUALIFIER allows any data definition, including procedures, tasks, and data items, to be stored in the library, it appears to perform the functions of both compools and libraries. Furthermore, selecting suitable names for sub-libraries and storing project oriented entities in them can help create project related libraries.

No conflict with other requirements.

---

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware. ....I

The language provides facilities for interfacing with the standard peripherals, the process peripherals, the sensors, and other hardware devices. These facilities, included in the system division of the program, are specified by the programmer for each specific installation. The problem division, containing the user's application program, is machine independent. The system division has to be modified for different computers or installations. It allows the user to specify the CPU connections with the devices by means of the user-specified channels. The SINGLE CONNECTION, the GROUP CONNECTION, and the TRANSFER DIRECTION allows specification of how any device exchanges data with any other device.

No conflict with other requirements.

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling. ....P+

Sequential execution .....T
Conditional execution .....T
Iteration .....T
Recursion .....F
(Pseudo) parallel processing .....T
Exception handling .....P-
Asynchronous interrupt handling .....T
Control structures from a small set of simple primitives .....F

The language provides for sequential execution, the IF and CASE statements for conditional execution, the REPEAT statement for iteration, several parallel processing features for creation, termination, scheduling, and synchronization of tasks. In the event-dependent scheduling of tasks, an event can be an interrupt and the language allows for WHEN-CONDITION and ENABLE, DISABLE/TRIGGER, or INDUCE options to handle these interrupts. Some exception conditions such as ON end-of-file and ON signal are provided for, although no provision is made in the language for many other exceptions such as overflow, underflow, zero divide, size, subscript range, string range, area, etc. The language does not provide for recursive procedures. Hence it only partially meets this requirement.

The language must provide for recursive procedures, preferably permiting specification of a limit on the depth of recursion. Exception handling ON conditions for various situations listed in the previous paragraph should be provided.

Implementation of these features in the existing compilers will require a moderate effort.

---

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition. ....P+

A GOTO statement does not permit a jump into a composed entity such as a task, procedure, BEGIN block, REPEAT statement, conditional statement, CASE statements, etc., since such entities are considered self-contained. An exit from the block-tail of a task or procedure using GOTO is also not permitted (pp. 77-78). However, exits from BEGIN blocks using GOTO are not explicitly prohibited, although the manual has no examples of such a use.

No conflict with other requirements.

---

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives. ....P

Based on Boolean expression .....T
Based on type from discriminated union .....F
Based on computed choice among labeled alternatives .....P
All alternative must be accounted for .....F
Simple mechanisms will be supplied for common cases .....T

The IF statement (p. 78) allows the control path be selected based upon the evaluation of a Boolean expression. The CASE statement permits a choice from among its labelled statements depending upon the computed value of an integer expression only. It also provides OUT and FIN statements for cases when the integer expression is negative or greater than the number of alternative branches, but the OUT statement is not required (p. 79). Simplified forms of these statements are also permitted. For example, an IF statement need not have ELSE clause associated with it.

The language does not support discriminated union at all.

The CASE statement must be elaborated to permit multiple branches based on values other than the value of an integer expression. As discriminated union is added to the language the IF and CASE statements should be modified to be able to use the tag field. Moderate changes will be required to meet this requirement.

---

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted). ....P+

Termination can occur anywhere in the loop .....T
Multiple terminatinn predicates are possible .....T

Entry permitted only at the loop head .....T
Simple cases are clear and efficient .....T
Control variable is local to the loop .....T
Control value is efficiently available after termination .....F

The REPEAT statement in PEARL comes close to meeting this requirement. It allows termination of the loop at any point by means of a GOTO or IF statement, it only allows entry to the loop at the top, and it permits simplification (e.g., it permits values of 1 to be substituted for expressions following FROM and BY if the user fails to provide a value and it has an optional WHILE clause for a common terminating case). It makes the control variable local to the loop and guarantees its validity within it. The language manual also specifies (p. 81) that the value of the control variable will be greater than the final value specified by the expression following the TO clause. However, that value is inaccessible. Indeed, because the loop variable is local to the loop, its value can only be accessed outside of the loop if it is explicitly assigned to some other variable before exit.

The language definition should specify that the value of the control variable be available to the user after the termination (normal or abnormal) of the loop. Implementation of this feature is trivial, the hardest part being finding an acceptable syntax.

---

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure. ....F

No recursive procedures within recursive procedures .....N/A
(Maximum depth of recursion can be specified) .....N/A
(Recursiveness must be specified) .....N/A

The language does not support recursion.

The recursive option must be supported by the language. The default should be no recursion so that the user does not pay the price for recursive code if he does not need it.

A moderate effort will be required to provide for appropriate code generation for recursive procedures and for related interactions. Limits on the levels of recursion permitted will keep the stacks and other problems to manageable sizes.

---

G6. The source language will provide a parallel processing capablity. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution. ....P

Able to create and terminate parallel processes .....T
Process can gain exclusive use of resources .....P
No parallel routines within recursive routines .....N/A
No routines within parallel routines .....T
Maximum number of simultaneous instances are declarable .....F
(Access rules are enforced) .....F

The language allows parallel processing capabilities. It permits the creation of a task by means of task declaration, its activation by means of the ACTIVATE statement, continuation by means of the CONTINUE statement, temporary suspension by means of the SUSPEND statement, and termination by means of the TERMINATE statement (pp. 133, 139-146). The task can only be declared at the module level. That is, there can be no task declaration within a procedure declaration, or within a task declaration. Thus subtasks are not permitted. There is no provision in the language to lock out files or records to prevent simultaneous access by two or more parallel tasks, but semaphore variables are available to simulate this capability in many cases. The language also does not allow specification of the maximum number of parallel tasks. Since the language does not permit recursion, the user cannot define parallel routines within recursive routines.

To fully meet this requirement the language should allow certain facilities to temporarily lock out a file or other resources for exclusive use by a certain task. The language should also allow the user to specify the maximum number of parallel tasks permissible at any one time.

The file handling capabilities of the language should be modified to include an 'EXCLUSIVE' option to allow a task exclusive use of that file until the resource is released. A moderate amount of effort will be needed to effectively implement these features in the existing compilers.

---

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program. ....P-

Program can get control for any exception .....P-
Parameters can be passed .....F
Can get out of any level of a nest of control .....F

Can handle the exception at any level of control .....F

The language does not provide for exception handling capabilities, except for the end-of-file capabilities. No provision is made to allow the user to control the flow of the program in case of overflow, underflow, zero divide, subscript range error, size error, etc. Exception handling parameters (e.g., labels) are not permitted to be passed in procedure calls.

The entire control structure associated with 'ON' conditions for error and exception handling must be added to the language. These structures should allow the user to terminate, resume or retry operations, permit constraints in flow of control, and in certain cases should permit exceptions by default.

A moderate to significant amount of effort will be required to include these features. These will affect the existing language syntax and code generation procedures.

---

68. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation. ....P+

Priority specification .....T
Synchronization via wait/enable operations .....T
Wait for end of real time interval .....T
Wait for end of simulated time interval .....F
Wait for hardware interrupt .....T
(Can enable and disable interrupts) .....T

The language almost meets this requirement, lacking only the ability to use simulated time. The task declaration includes a specification of PRIORITY and permits a task to wait until the completion of an event or a time duration before reactivating. Time-related events can occur at specified times or after specified intervals. The language also permits ENABLE and DISABLE options to enable or disable or induce certain interrupts (pp. 133-150). Special semaphore variables are provided to allow task synchronization when used with the REQUEST and RELEASE operations.

No conflict with other requirements.

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous. ....P+

Free format with statement terminator .....P+
Mnemonic identifiers possible .....T
Based on conventional forms .....T
Simple grammar .....T
No special case notations .....T
No abbreviations of identifiers or keywords .....F
Unambiguous grammar .....P+

The PEARL language permits free formatting of its statements and requires a semi-colon to terminate them. The only exception to the rule is that blanks are not allowed in the composite operators (e.g., ==, >=, ** etc). The language allows the user to specify mnemonic names although it permits the compilers to scan only the first six alphanumeric characters to establish the uniqueness of the identifiers. The language constructs are in general readable and permit a left to right scan of statements. Rules for expression evaluation, parameter passing and other salient features of the language are simple and easily understood. There are no special language constructs to support rare cases. The language, however, permits abbreviations for some of its keywords. For instance the abbreviations DVC, SPC, DCL, INIT and PRIO are permitted to represent DEVICE, SPECIFY, DECLARE, INITIAL, and PRIORITY respectively. Furthermore the language provides clear syntax with few exceptions. For example, the constructs RETURN and RETURNS are syntactically very close to each other, yet have two separate and distinct functions to perform. One is used to return the value of the function while the other is used to define the mode of the function value. Two distant syntax entities should be used to perform these functions.

Minor modifications to the language syntax are needed to fulfill this requirement. Abbreviations for keywords should be forbidden. Those keywords, such as RETURN and RETURNS, which are syntactically very similar and yet perform two different functions, should be altered and made syntactically distant.

Inclusion of these modifications in the existing compilers is trivial. It may impact the user programs more severely.

---

H2. The user will not be able to modify the source language

syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences. ....I

The language definition, its features and rules do not allow the user any means to alter the language syntax. He cannot alter operator hierarchies, precedence rules, define new keywords or infix operations. The most he is allowed to change is a few special characters which are installation dependent. This may alter parts of his programs, but does not affect the language defined constructs.

No conflict with other requirements.

---

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset. ....P

All language constructs are composed of a character set consisting of 26 alphabetic characters (A-Z), 10 numeric characters (0-9), and 15 special characters, which are all part of the ASCII character set. The language does, however, provide for installation defined special characters, some or all of which may not be formed from the 64 character ASCII set. (p. 31)

A further constraint should be placed on the installation dependent character set requiring that no special characters that cannot be formed from the standard 64 character ASCII set should be permitted in the set. Even better would be to forbid any installation dependent characters.

Implementation of this constraint is trivial.

---

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals. ....P

Break character exists .....F
(Literals are self-identifying as to type) .....I
(Bit-string literals for any type) .....F

The language provides the formation rules for identifiers and literals (pp. 32-38). It does not, however, provide for a break character which can be used to increase the readability of the identifiers. In fact, it explicitly forbids the use of blanks and special characters for use within identifiers.

The formation rules for identifiers and literals should be modified to include the use of certain language specified break characters. This would increase the readability of language constructs. Perhaps the number of characters to establish uniqueness of the identifiers should also be increased from 6 to at least 12.

Implementation of these suggested changes to the language will affect most existing translators but its scope will be minor.

---

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings. ....F

Nowhere does the language manual specifically permit identifiers, constants, etc., to be continued across lines. However, the manual explicitly states that comments or its portions can be carried across lines, implying that the same rule is permissible to lexical entities also. (p. 37)

The manual should explicitly forbid the continuation of lexical entities across lines.

Minimal changes to existing implementations will be required to accommodate this language modification.

---

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used. ....P

The keywords in the language are reserved and cannot be used as identifiers. However, the number of keywords is not small (152). As a rule these keywords are informative. (See appendix A2.3.)

Part of the reason for such a large number of keywords is that many of the keywords have abbreviations which are also treated as keywords. If the abbreviations were eliminated, as suggested in requirement H1, the list of keywords can be significantly reduced.

Minimal effort is requires to implement this change.

---

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs. ....P+

Uniform comment convention .....T
Look different from code .....T
Bracketed by one or two characters .....T
Can contain any characters .....T
Can appear anywhere reasonable .....T
Terminated by the end of the line .....F
Compatible with automatic reformatting .....T

The language allows a single uniform comment convention. These comments can be easily inserted into the program text between language elements. They must be delimited by /* and */. Any single character can be included in the comment, only the */ combination is excluded since it signifies termination of the comment. The language definition, however, permits continuation of comments over several lines and does not require their termination at the end of the line. These comment conventions do not prohibit automatic reformatting of programs. (pp. 37-38)

To meet this requirement the language must require that the comment terminate at the end of line in addition to terminating at */ combination.

Minimal implementation changes are required to incorporate this language change.

---

H8. The language will not permit unmatched parentheses of any kind. ....T

PEARL fully meets this requirement.

---

H9. There will be a uniform referent notation. ....P+

Function references and array element references have similar notation in PEARL, the arguments being enclosed in parentheses. However, assignment to a function is not possible. Since functions cannot return structure values, the question of dot qualification of a function name does not arise.

To allow assignment to a function would be a moderate effort.

---

H10. No language defined symbols appearing in the same context will have essentially different meanings. ....I

The language assigns unique meaning to its symbols. The usage of its symbols is unambiguous. For example the symbol = is used for assignment while == is used for logical equality comparisons. There is, however, redundancy in symbols. Both := and = are used for assignment.

The redundancy in the language operators should be eliminated by simply disallowing one symbol to be used in the language. This will increase the efficiency of the language.

Minimal implementation changes will be required in the syntax checking mechanism and the symbol table to affect this language modification.

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program. ....P

In many instances the defaults are language defined. For instance, if a procedure is not explicitly declared REENTRANT, the automatic default is that it will be treated like a non-reentrant procedure which will not be callable simultaneously from two or more tasks. Similarly, in the SEND or TAKE statements if the user fails to specify the SYMBOL (the location of buffer) option, the default is assumed to be the storage cell or array assigned to the I/O device. However, in many other instances the language fails to define the default condition and leaves it implementation dependent. For example, no exception handling mechanisms are provided in the language. Hence, the action taken or the flow of control after an overflow, underflow, size error, subscript range error, etc., is implementation dependent. Similarly the language does not provide lockout facilities for file control. In parallel processing situations when two or more tasks vie for the same file at the same time, what happens is undefined. The language fails to specify ways to prevent such deadlock.

The language must carefully specify a means for handling exception conditions or list the default actions that compilers must take when exception situations arise. In parallel processing also, file lockout and other mechanisms should be provided to avoid deadlocks or simultaneous access of the same file by two or more tasks when that is not desired.

Implementation of these facilities will require moderate additions to existing compilers. The syntax handling routines, task handling routines, and I/O handling routines will have to be modified. Exception handling capabilities will have to be added.

---

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary. ....P

Defaults specified for don't care cases .....T
Programmer can override the defaults .....F

The language permits a large number of defaults to be taken care of by implementations. These are "don't care" types of defaults. They

include such cases in which a procedure is treated as non-reentrant if the REENTRANT option is not provided, the size of an input record in stream type of input, etc. But other types of defaults which affect the result or the flow of control of the program are not "don't care" type of defaults. This category includes provisions for exception handling, subscript range checking, etc. The language does not allow the user the facility to override these types of defaults.

The changes necessary to give an ability to override these exceptional defaults are discussed under requirement 67.

---

I3. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration. ....I

PEARL provides the user the facility to specify the configuration characteristics of the hardware system necessary to execute his problem program. This is done by means of the system division in which the user describes the hardware configuration, the standard peripherals, the process peripherals, the device connections, the sensors, interrupts, and signals. PEARL enables the programmer to declare problem specific designators for devices and device connections which can then readily be used in the problem division of his program. Thus a meaningful declaration of the process dependent data is guaranteed. Another advantage of this facility is that whenever changes in the hardware configurations have to be made, they can be restricted only to the system division without modification to the problem division. Furthermore, the system division of a PEARL program provides for the automatic generation of a problem specific executive system.

No conflict with other requirements.

---

I4. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled. ....F

PEARL has no conditional compilation capability.

The language designer must consider providing for such conditional compilation facilities. They should also establish as to where in the program the user must specify conditional expressions which are compile time evaluable. The result of the evaluation will then decide which of the several environments that are possible for the program should be generated.

This facility can usually be implemented in most compilers at a modest cost, the cost depending primarily on the source level elaborateness of the facility. The required changes usually can be isolated in the initial phase of the compiler.

---

I5. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language. ....U

The language manual does not recognize nor clearly specify the avionic subset of the PEARL language as the kernel of the overall PEARL language. The preface to the manual states that there are commonalities between 'avionic subset' and 'PEARL-Basis-Subset'. However, a list of those commonalities which may possibly constitute the kernel of the language is not available. Furthermore, to qualify as the kernel of the language, these 'commonalities' of the language must be sufficient so that their implementation alone will make the full source language capability available through extensions.

The kernel of the language which contains all the power of the language must be identified. The other language capabilities should be definable and implementable in terms of the features of the kernel.

A moderate amount of effort will be necessary to define the kernel of the language if one does not exist at present.

---

I6. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition. ....P

The language specifies few limits on its features which are basically machine independent. For instance, it requires that identifiers be 6 alphanumeric characters long. However, it does not specify limits on several other language constructs. For instance, there is no language specified limit on the number of dimensions of an array, on the maximum number of REPEATs within REPEATs, IFs within IFs, the maximum number of identifiers in a procedure or program, the maximum number of statements in a procedure, maximum number of formal and actual parameters, etc. Hence, it only partially meets this requirement.

To implement this change would require analysis of user requirements and establishment of limits for various features, such as the ones listed in the previous paragraph, so that these limits will not be exceeded by any application. The effect of such standardized limitations on existing implementations which meet or exceed them will be minimal, but the effect on other implementations could be heavy.

---

17. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator. ....T

The language does not contain restrictions which have been imposed by the hardware or object environment with the possible exception of CLOCK data type and related features which require access to a real-time clock. However, this is an essential feature to support the real-time capabilities of the language. The language does not put limits on run-time storage requirements or the type or number of peripherals or other special hardware devices.

No conflict with other requirements.

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs. ....P

No efficiency cost for unused features .....P
Efficient code can be produced for all features .....P

There are some features of the language which may be costly in terms of run time or code even when not used. Procedure calls are one such example. There is no provision for a NORECALL option for procedures which can save execution time. The user is not allowed the option of OPEN or CLOSED procedures and is thus forced to pay for closed calls.

Modifications to language features to include OPEN, CLOSED, NORECALL and similar other optimizing features in the language will increase the compilers' capability to generate efficient code. Implementation of these capabilities will require a moderate amount of effort in the code generation and syntax analysis portions of the compiler.

---

J2. Any optimizations performed by the translator will not change the effect of the program. ....U

This is basically a translator requirement. However, certain features of the language, such as the REENTRANT option, permit the translators to generate non-reentrant code in their absence. It permits generation of more efficient code without changing the effect of the program.

The requirement is more applicable to the translators than to the languages. Languages can, however, provide certain features (e.g., OPEN, NORECALL, PACKING of data etc.) which can permit optimization of space and/or time.

---

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions. ....P

The system division of PFARL provides encapsulated access to machine dependent hardware facilities. The user has control of device

management and connections. He can control I/O channels and associate them with desired peripheral devices. He can also handle certain hardware interrupts and signals. He can address these devices, interrupts, etc., in the problem division. (pp. 100-109)

However, PEARL does not specify interfaces with machine code in any way. This interface is either not defined or can be achieved by a procedure call to a machine language procedure. The language manual does not specify that description of machine characteristics for which the machine code is being introduced be present in the code.

The language manual should clearly state how the interfaces with the assembly language procedures are to be established including the details of the format for such a call.

---

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator. ....F

Encapsulated specification of representation possible .....F
Space/time tradeoff can be specified .....F

The language does not allow packing factors to be specified with the data structures, nor does it allow the length of each field to be specified in the structure. The language does, however, permit length specifications for fixed point and floating point data. No facility in the language is provided to specify a space/time tradeoff to the compiler. (pp. 54)

The language data structure definition should allow packing of data as well as the user specification of field length. Other space/time tradeoff directives should also be included in the language which will permit the compiler to try to optimize for space or run-time wherever possible.

The existing translators will have to be modified to accept and process the language space/time related directives. They will have to be modified to provide for new data structure definitions.

---

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics. ....F

Open/closed properties can be specified .....F
Open and closed versions have the same semantics .....F

The language does not provide this option to the user.

The procedure definition should be modified to permit OPEN or CLOSED option to be provided by the user. All existing implementations will have to be modified to include this option. The cost will be moderate to high.

## Extraneous Features

We recommend that the following features of PEARL, not required by the Tinman, be kept:

* CHAR (character string) type and the CAT (concatenation) operator (assuming that the Tinman requires only a character type, not a character string type).

* CLOCK, DUR, DEVICE, FILE, INTERRUPT, SIGNAL, and SEMA types. Although not explicitly called for by the Tinman as types, they are needed in the PEARL realization of other Tinman requirements (e.g., parallel processing, device independence).

* Call by value parameters.

We recommend that the following features of PEARL, not required by the Tinman, be deleted:

* The LENGTH global specification and data attribute. These functions will be assumed by range and precision specifications in any extension of PEARL to meet the Tinman requirements.

* The FIT operator. The function of and need for this operator (which "fits" an integer of one length to another length) are unclear, but it appears that if range specification for integer data is added to the language, as called for by the Tinman requirements, then it will not be needed because explicit conversion between ranges is unnecessary, according to requirement B8.

* The SKIP statement. This is a high-level no-op and PEARL, unlike FORTRAN, has no need for such a statement. It could functionally be replaced by a comment.

The two shift operators (SHIFT and CSHIFT) perform a useful function, but their uses are almost always hardware dependent. Therefore any use should be bracketed (encapsulated) by statements which call attention to that dependency. This is particularly true for CSHIFT (circular shift), which is often implemented in a highly dependent fashion. (It is difficult to functionally simulate a circular shift in a datum which occupies only part of a hardware register.)

## Summary

PEARL supports several of the real-time capabilities called for in the Tinman completely or partially. It provides for input/output capabilities. The user has control of channel programming, device assignment, interrupt handling, and signal handling. He can access and manipulate files. He can perform graphic input or output. The language also provides for parallel processing features. There are language constructs for multiple task creation, termination, resumption, synchronization, and continuation. The user can assign priorities to his tasks. He can establish communication between these tasks. The language also supports real-time processing. Two data types (CLOCK and DUR) are provided in the language to account for real-time and time duration. The tasking facility is combined with real-time options to start, delay, or terminate tasks at certain appointed times or after certain time durations.

In addition, the language supports many other Tinman requirements. It requires explicit declaration of type for all entities at the beginning of their scope and before their use. It provides for numeric, character, clock, duration, array, and structure data types. It allows explicit mode conversions between different data types by providing conversion operators (e.g. FIX, FLOAT, CHAR, FIT, etc.). It allows global specification of the length and mantissa for fixed point and floating point variables. It requires that the number of dimensions of arrays, their type, and their lower bound be fixed at compile time, although it permits dynamic arrays whose upper bounds can be determined at scope entry time. It allows establishment of equivalence between variables by means of the IDENT attribute. The language syntax permits differentiation between assignment and logical equivalence and provides for various logical comparison operations. Checking for type and number of actual and formal parameters is required in the language definition, as are calls by value and reference. The language allows an identifier to stand for a constant. It provides for data initialization as part of data declarations and does not permit declarations by default. Additionally, the language provides a special capability for specifying the name of a library into which the files or procedures are to be stored.

Despite these characteristics that the language possesses, there are others in which the language is deficient. It does not have a fixed point data type. It does not provide for user-defined data types or operators, nor does it allow extension of existing operators to new data types. The language also does not support the notion of defining new data types by enumeration or as Cartesian products, discriminated unions, or powersets. Another significant deficiency in the language is the absence of any kind of pointer mechanism which can enable a user to manipulate lists and recursive data types. Recursion at procedure level is also not supported in the language. Furthermore, there are no exception handling control mechanisms in the language to control subscript range error, size error, overflow, underflow, etc. No less significant is the lack of a well defined interface with routines

compiled from assembly language and other high order languages. The language also fails to provide for mechanisms to control compilations to generate code for specific hardware environments.

In addition to the major omissions listed in the previous paragraph, the language fails to meet the Tinman requirements in some other ways. It does not provide for precision and range specification of variables at the time of their definition. There is no provision for the operation of integer division with remainder. The language does not require that all truncation during compilation take place only from the least significant bits on the right. Scalar operations on compatible arrays and records as a whole are not defined. Constant expressions cannot replace constants anywhere in the program, in particular in the declarations. The language does not require that the value of the control variable in a loop be available after normal or abnormal termination of the loop. Several keywords have alternate abbreviated forms. There is no language defined break character, and the language does not provide the user the option to specify open procedures.

Although a significant amount of effort will be required to modify the language definition and syntax and to extend the capabilities of the present definition of the language to include those features in the language required by Tinman, we believe that such an effort can be successful. With the excellent I/O, tasking, real-time and hardware interface capabilities that the language currently possesses, an extended language to meet the future DOD requirements can be built. Portions of the existing syntax will have to be modified, others eliminated and some new ones added. However, the language does contain enough substance and character to provide the basis for such a modification and extension.

# A COMPARISON OF
# SPL/I
# to
# TINMAN

*Final Version*

31 December 1976

PREPARED BY
*COMPUTER SCIENCES CORPORATION*

# Introduction

This report gives a comparison of the language SPL/I to the Tinman language requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1976, Section IV). For the purposes of this comparison, SPL/I is considered to be defined by:

SPL/I Language Reference Manual for Compiler Release 4.0
Intermetrics, Inc.
Cambridge, Mass.
July, 1976

Tinman contains 79 language requirements. This report compares SPL/I to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used. (Occasionally no text is needed if a requirement is totally satisfied.) If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - Only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

(The symbols P, P+, and P- will often be used with requirements which

are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

The report concludes with two summaries. The first is of the features of SPL/I which are extraneous to Tinman and the desirability of retaining each of them. The second is of the language as a whole and the desirability of modifying it to bring in into line with the Tinman requirements.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs. ....T

The SPL/I word for "type" is "mode". The requirement is met (see, in particular, p. 49). The modes of formal parameters must also be given (p. 79).

---

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators. ....T

Integer .....T
Floating Point .....T
Fixed Point .....P-
Boolean .....T
Character String .....T
Arrays .....T
Records .....T

SPL/I has all required types except fixed point; it does have a fraction type as a partial substitute. The following additional types are provided as well: Complex integer, complex fraction, complex floating point, double precision integer, signal, resources, process, bit. (See pages 49-68.)

Adding fixed-point would be of moderate difficulty. See also the comments under A5 on the character type.

---

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code. ....T

Global arithmetic precision specification mandatory .....F
Individual variable precision specification permitted .....F

The precision of floating point arithmetic and variables is implementation dependent (p. 52).

A moderate change would be necessary to meet the requirement. The change does not strongly interact with other language features, except that a consistent syntax should be designed to specify properties of variables of all types. Routines to perform the arithmetic to the necessary precision would be needed; note that SPL/I does not even have double precision floating point capabilities.

---

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler. ....F

Treated as exact quantities .....F
Range and step size determined at compile time .....F
Scaling handled automatically .....F

SPL/I has no fixed point type.

(See the comments on A2.)

---

A5. Character sets will be treated as any other enumeration type. ....F

New sets can be defined as enumeration types .....F
ASCII and EBCDIC are provided .....F
(Conversion capability between sets is available) .....F

SPL/I has only one character set, and the internal representation is implementation dependent. (p. 53)

The necessary change is not hard; nor is it trivial. SPL/I does not even have an enumeration type generation capability, so both this and the ability to define new character sets would have to be added together. Moreover, it would be desirable to implement a routine to convert a character string from one character set to another.

---

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope. ....P

Number of dimensions is fixed at compile time .....I
Type is fixed at compile time .....I
Lower subscript bound is fixed at compile time .....F
Upper subscript bound is fixed at scope entry .....F
Subscripts only integers or from an enumeration type .....?
Subscripts will be from a contiguous range .....I

The lower subscript bound is fixed by the language definition; it is always 1 (p. 116). The upper subscript bound is fixed at compile time (p. 64). Subscripts may only be integers; SPL/I has no enumeration types (p. 116). The other requirements are met (see p. 64-66, 116-117).

Some of the required properties are available by means of array slice references (p. 118-121); using these to obtain variable bounds is quite inefficient, however.

Assuming that enumeration types are added in any case, the change necessary here would be moderate. Even allowing only the upper bound to be fixed at run time adds a level of indirectness for references to all but one dynamic array per scope, and interfaces to the dynamic stack allocation mechanism would have to be added.

---

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time. ....F

Alternative structures for records are possible .....F
Discrimination condition may be any Boolean expression .....F

SPL/I does not provide for alternate structures for records, which are known as "structures" (pp. 62-63).

The necessary change would be of easy to moderate difficulty. Some redesign of the case statement would be required (or at least highly desirable), to permit discrimination based on the structure -- case

labels are currently only integers. If run-time enforcement checks were wanted, more work would be necessary, although SPL/I does require full qualification for component names (p. 122).

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value. ....P+

Automatically defined for any type (except...) .....P
Available for individual components .....T
(Assignment and reference via functions) .....F

The intent of the requirement is fully met for all types which SPL/I has -- it has no user defined types (pp. 157-158).

However, variables of certain types cannot be assigned to in an assignment statement, for good reasons (e.g., the PROCESS type; see pp. 165-167).

The only necessary change is in the wording of the Tinman requirement. The intent cannot be to require assignment and reference for every type, for some it is not meaningful or not safe.

---

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity. ....P+

The '=' and '/=' operators can be used with any type. However, both operands must be of the same type. (p. 96-97)

The necessary change would be very minor.

---

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration. ....P+

Built-in for all numeric and enumeration types .....P+
Ordering can be inhibited when desired .....T

The SPL/I relational operators are defined for all numeric types; the language has no enumeration types, or unordered types to which relationals can be applied. (pp. 96-97)

The only necessary change is to add enumeration types; see F6.

---

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation. ....P+

Addition .....T
Subtraction .....T
Multiplication .....T
Division with real result .....T
Exponentiation .....T
Integer and fixed point division with remainder .....P+
Negation .....T

The remainder from an integer division is not accessible, but there is a MOD function. (pp. 102-103, and Appendix B, pp. B-5 and B-6).

The result of a division is of the same type as the operands, so explicit conversions are necessary to get a real result from the division of one integer by another. This is not a violation of the Tinman, but is rather unusual.

The necessary change is trivial.

---

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers. ....U

Never from the left for data within range .....U
Never on the right for integer and fixed point .....U
Implicit floating point rounding beyond precision allowed .....U
(Run time checks can be avoided) .....U

Nothing is said about the ranges of results, or about truncation.

---

B6. The built-in Boolean operations will include "and", "or", "not", and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode. ....P+

Short-circuit and .....P
Short-circuit or .....P
Not .....T
Xor .....T

All the required operators are present, but AND and OR are not necessarily evaluated in short-circuit mode. (pp. 87-91)

The required (and questionable) change is of moderate difficulty. The rules for evaluation order in expressions would have to be changed.

---

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure. ....T

Scalar operations on arrays .....T
Assignment between records and arrays of conformable type .....T

Component by component operations on arrays are provided (pp. 91-92). They are apparently not provided for records (p. 92). Scalars are considered compatible with arrays having components of the same type (p. 102), and scalar actual parameters are even compatible with array formal parameters (p. 80).

Assignment also meets the requirement (pp. 157-158).

---

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors. ....P

Requirement B8

No implicit conversions .....T
Explicit between integer, fixed point, and floating point .....P
Explicit between fixed point scale factors .....F
(Explicit between integer and Boolean) .....T
(Explicit between integer and enumerated types) .....F
(Explicit between different enumerated types) .....F

SPL/I has no fixed point. However, conversions between integer and floating point, and floating point and fraction, are provided. (pp. B-22 through B-24). Many other explicit conversions are also provided, including between BIT and Boolean, and BIT and integer, and character string and integer, floating point, and fraction (pp. B-22 through B-32).

There are no implicit conversions - not even mixed mode arithmetic (pp. 87-103).

The necessary change is to add fixed point, see A2.

---

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated. ....P-

Implicit conversion between ranges .....F
Exception condition on integer and fixed point truncation .....P

The only instance of range in the language is integer and double precision integer, and as these are considered different types, explicit conversion is required (p. 102).

Run-time exception conditions on truncation are mentioned only for STRING and BIT assignment (p. 159).

Providing for run-time exception conditions on truncation would be of moderate to high difficulty; the language has no mechanisms in this area at all.

Adding ranges (and implicit conversions) would be of moderate difficulty.

---

B10. The base language will provide operations allowing

programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent. ....P

Sending and receiving of data .....T
Sending and receiving of control information .....T
Dynamic device assignment .....F
User exception condition control .....F
Installation independence .....F
(Data formatting capability) .....F
(Reading and writing of bit strings) .....T

There are only six I/O procedures in SPL/I: OPEN, CLOSE, PUT, GET, WRITELINE, and READLINE. They operate with virtual channels, and are installation and device dependent. Dynamic device reassignment is not provided. (pp. B-16 through B-21)

Just what the Tinman requires is not at all clear enough in detail to determine whether the necessary changes are fairly easy or moderately difficult.

---

B11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate. ....P

Union .....T
Intersection .....T
Difference .....F
Complement .....T
Membership predicate .....F
(Set inclusion) .....F

SPL/I has no powersets, but some of the required operations are provided on bit strings, which are in effect powersets where the elements are unknown to the compiler. (pp. 90-95)

It would be easy to add a difference operation. Adding a membership operation would require also adding powersets, which would take somewhat more work (but less than adding fixed point, for example).

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right. ....F

Side effects must occur in left-to-right order .....F
(Embedded assignments) .....F

The SPL/I rule is that the order of evaluation of operands of a single infix operator is undefined. This is a much better rule than the Tinman requirement, but it does not meet the requirement. (p. 97)

The change is moderately difficult. It would have a major effect on code generation in the compiler, and would lead to worse code in a number of cases.

---

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized. ....F+

Few precedence levels .....I
No user-defined precedence levels .....I
Operands of an operation are obvious .....P

There are nine precedence levels, which seems reasonable (p. 98). There is no way to define new operators, much less new precedence.

Operands of an operation are usually obvious, but A/B*C and A/B/C are permitted (p. 100).

Disallowing the two noted cases should be relatively easy.

---

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed. ....I

Fully met (pp. 87-129).

---

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time. ....P+

In a few contexts, only integer literals are allowed -- for example, the repetition factor within a structured constant (p. 111), and the labels on parts of a case statement (p. 138). Array upper bounds can however be expressions, evaluable at compile time (p. 64).

The required change would be fairly easy, although some care would be necessary to avoid syntactic ambiguity.

---

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to ease of learning. ....T

Parameter rules consistent in all contexts .....T
No special operations applicable only to parameters .....T

SPL/I really uses parameters only for procedure calls, so the question of consistency does not arise. Moreover, a procedure name cannot be passed as a parameter (pp. 51, 67, 79, 154), so no additional ways of binding arise for parameters.

---

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter. ....P+

Actual and formal parameters will agree in type .....T
Rank of parameter arrays is fixed at compile time .....T
Parameter array size and subscript range can be passed .....P+

The only violation is that lower subscript bounds cannot be passed, as they are fixed at 1 by the language (p. 114). Note, however, that an array slice (pp. 118-121) can be passed as an array, to get some of the capability.

Binding of formal to actual parameters is discussed on pp. 79-81.

The necessary change is part of the work of adding lower bounds other than 1; see the comments on requirement A6.

---

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters. ....P

Act as constants (call by value plus) .....T
Act as variables (call by reference) .....T
Exception control .....F
Procedure parameters .....F
(Act as variables, but call by value) .....T
(Act as variables, result parameter) .....F

Both VALUE and INPUT parameters are provided. The first is the same as Algol 60 (the actual parameter does not change, but the formal parameter may); the second is the constant parameter called for in the Tinman. Call by reference (renaming) parameters are called INOUT.

There are no special provisions for exception control parameters, and procedure names cannot be parameters. (pp. 51, 67, 79-81, 154).

The required extensions would be moderately difficult, about the same amount of work as adding enumeration types. The missing features do not interact strongly with other language features.

---

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time. ....F

Above properties optional .....P-
Above properties are fixed at run time .....F

Everything that can be specified about a formal parameter -- i.e., its mode and size -- must be specified. (pp. 67, 79). The concept of generic procedures is not in the language.

Note that if an array is passed through two levels of procedures, its size must be connected to the array at each level, even if the array is referenced only as an actual parameter at that level.

The addition of generic procedures would be fairly hard, particularly if efficient code is desired.

---

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time. ....P-

Variable number of arguments possible .....P-
All but a constant number of arguments have the same type .....F
Number of arguments in each call is fixed at compile time .....T

Only the built-in GET and PUT I/O procedures accept a variable number of arguments, and nothing is said about type. (pp. 79-81, Appendix B)

The change would be at least moderately difficult.

D1. The user will have the ability to associate constant values of any type with identifiers. ....F

There is no way to associate an identifier with a constant in SPL/I. (p. 49)

The required change is trivial.

---

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output). ....P+

Literals for all built-in types .....T
Consistent interpretation in program and data .....U

Literals are provided not only for all built-in data types, but for generated types (e.g., arrays, records) as well, in a way which is easy to read. (pp. 107-113)

Nothing is said about consistency of interpretation for programs and data, but since precision is implementation dependent (p. 52), we doubt that the requirement would be met by cross-compilers.

Providing precision in a machine independent way would be moderately difficult. See also D3.

---

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values. ....P+

Initial value can be specified as part of the declaration .....T
Initialization occurs at allocation scope entry .....T
No default initial values .....U

The requirement is probably fully met, but nothing is said about default initial values, and reference to a variable without a value is apparently not detected at run time. (See pp. 69-71, 114-123.)

---

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types. ....F

Numeric variable range specification mandatory .....F
Fixed point variable step size specification mandatory .....F
Range and step size specifications do not define a new type .....F

SPL/1 has no way to declare range or step size, and integer and double precision integer are treated as two different types, with explicit conversion necessary to assign one to the other. (See variable declarations, pp. 67-71; primitive modes, pp. 51-54; and assignment, pp. 157-159).

The necessary change is moderately difficult. The syntax needs to be changed to allow for range and step size declarations, and compile-time and run-time checks must be implemented. There is a strong interaction between this and the addition of fixed point.

---

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a contiguous subsequence of any enumeration type. ....F+

Ranges of an enumeration type are allowed .....F
No arbitrary restrictions on the structure of data .....T

There are no enumeration types, ergo no ranges thereof (pp. 49-68).

There are no arbitrary restrictions on the structure of data; records can be components of arrays and vice versa (pp. 49, 62-66). It is stated on page 117 that an array element is a scalar, but this is clearly a simple wording error in the manual.

Once enumeration types are added (see E6), and integer ranges are added (see D4), providing ranges of enumeration types would be trivial.

---

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables. ....F

Recursive and network structures provided .....T
Handles variable-value and structure-component connections .....F
Pointer property is an attribute of a typed variable .....T
Pointer property not for constants, affects only assignment .....P
Pointer property mandatory for dynamic allocation .....N/A
Allocation scope never wider than access scope .....T
(Either the value or the pointer is modifiable) .....T
(Pointer mechanism handles procedures and parameters) .....F
(Remap and replace assignment have different syntaxes) .....T
(Built-in dynamic variable creation) .....T
(Variable equivalence classes are declarable) .....F

SPL/I's pointer mechanism is almost identical to that of Pascal; in other words, it is a good one. POINTER is a mode, but pointer values are not directly accessible. A pointer variable points to an object of a specified type. Either the pointer or the pointed-to value can be assigned to. The latter is referenced as, e.g., p$ (similar to Pascal's p^). A pointer can share its pointed-to value only with another pointer variable. A heap allocation mechanism is provided. (pp. 59-61, 128-129, B-10, B-11)

To change SPL/I's pointer mechanism to meet the Tinman requirement would be a lot of work. Much of the work would be in defining the requirement itself to be a viable, sensible one, in more detail than at present.

E1. The user of the language will be able to define new data types and operations within programs. ....P-

No special mechanisms are provided for defining new types. However, arrays and records are provided as type generators, and own variables (called STATIC) would allow one to define and use a new type via procedure calls, with the components of the data objects inaccessible except via the procedures. This provides much of the required capability, albeit inefficiently, even though the methods are perhaps different from those envisioned by the Tinman.

Addition of a special purpose abstract type mechanism would be a major effort.

---

E2. The "use" of defined types will be indistinguishable from built-in types. ....F

See the comments on requirement E1.

---

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations. ....T

Fully met (see pp. 36-40).

---

E4. The user will be able, within the source language, to extend existing operators to new data types. ....F

See the comments on requirement E1.

---

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the

type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented. ....P-

Construction .....T
Selection .....F
Predicates .....F
Type conversions .....T
Operations and data can be defined together .....P

The only way to define operations on new types is by procedures. This allows construction and type conversion operations, with a notation much like that for built-in types. It does not allow the definition of infix predicate operators, or of selection operators which can appear on the left hand side of an assignment statement.

See the comments on requirement E1; a major change would be required.

---

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time. ....P

Enumeration .....F
Cartesian products (records) .....T
Discriminated union .....F
Powerset of an enumeration type .....F

Only pointers, records, and arrays are provided as type generators. (pp. 58-66)

A moderate change would be required. Adding enumeration types and powersets would not be too hard; most powerset operators are already provided (see 8-11). Adding discriminated union would be somewhat harder, as it would affect the syntax and semantics of more statements (e.g., case).

---

E7. Type definitions by free union (i.e., union of

non-disjoint types) and subsetting are not desired. ....I

Only pointers, records and arrays are provided as type generators. (pp. 58-66)

---

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type. ....E-

Initialization .....I
Finalization .....E
Allocation actions .....I
Deallocation actions .....I

Allocation could be defined as a procedure, using either a large pre-declared array or heap storage via pointers. Initialization could also be defined as a procedure, but with a notation considerably different from that for built-in types. Finalization and deallocation cannot be defined in a sensible way, because there is no way for a procedure to automatically get control at exit from a scope.

See the comments on requirement E1.

R1. The language will allow the user to distinguish between scope of allocation and scope of access. ....P

Allocation and access scope can be different, but only if the allocation scope is the whole program, as in Algol 60 or CS-4 (pp. 69-71).

The change is fairly easy. The AUTOMATIC storage class declaration would need an optional parameter, giving the block name of the allocation scope block.

---

R2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components. ....P

Allowable operations can be limited .....T
Access can be limited where used .....T
External declarations need not all have the same scope .....F
Naming conflicts can be avoided (renaming) .....F

Access can be limited where a structure is used by only declaring as EXTERNAL the desired structures. There is, however, only a single external scope (GLOBAL on definition, EXTERNAL on reference). Allowable operations can be limited by making the data objects own (STATIC) within a procedure defining the operations. There are no re-naming provisions. (pp. 69-71)

Satisfying the requirement with a special purpose "cluster" mechanism would be a fairly major change, closely connected to that required by R1.

---

R3. The scope of identifiers will be wholly determined at compile time. ....T

Fully met (pp. 36-40).

---

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language. ....F

The run-time support library contains no such procedure (Appendix 3).

Once abstract types are added (requirement F1), then meeting this requirement could mean only a little work or a tremendous amount, depending on the size of the desired library.

---

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages. ....P-

Program component libraries accessible at compile time .....F
Libraries can contain foreign language routines .....F
Interface requirements checkable at compile time .....I

Compile-time libraries are not mentioned in the Reference Manual. It is, however, possible to declare the properties of separate compilation modules. (pp. 33-35)

The necessary changes are at least moderate in scope.

---

F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program. ....U

Libraries and compools will be indistinguishable .....U
Immediately accessible sublibraries at any level .....U

There is no mention of library mechanisms in the Reference Manual. (The requirement is not really a language requirement).

---

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware. ....F

While the multiprocessing and signalling statements (pp. 159-196) might be usable to partly meet this requirement, their connection to real hardware is implementation dependent (p. 187).

The requirement is so vague that it is impossible to estimate the scope of the necessary change.

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling. ....P+

Sequential execution .....T
Conditional execution .....T
Iteration .....T
Recursion .....T
(Pseudo) parallel processing .....T
Exception handling .....P-
Asynchronous interrupt handling .....P-
Control structures from a small set of simple primitives .....T

While the parallel processing facilities of the language could presumably be made to handle interrupts, the association of hardware signals with software signal variables is not part of the language definition (p. 186). (The language does specify a syntax for associating both hardware and software signals with signal variables, but the semantics of the association are implementation-dependent.)

There are no mechanisms for handling specific exceptions (the same implementation-dependent technique mentioned in the previous paragraph is available, but each implementation will specify which exceptions can be handled), and neither a label nor a procedure name can be passed as a parameter (pp. 90-104, 124, 126, 186).

Control structures are described in pp. 131-196.

Only a moderate change would be required. Much of the necessary mechanism is already present in the parallel processing facilities.

---

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition. ....I

Fully met. (p. 156)

While the requirement is met, the restrictions on GOTO and on ENDLOOP, EXIT, and ENDITERATION statements will lead to very awkward programming in some cases. For example, ENDLOOP cannot lead to outside of a BEGIN block in the iteration statement; EXIT must be used (and in most cases, ENDLOOP later). (pp. 147-150)

An unrestricted GOTO, used carefully, would lead to more readable, more efficient programs.

---

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives. ....P+

Based on Boolean expression .....I
Based on type from discriminated union .....F
Based on computed choice among labeled alternatives .....I
All alternative must be accounted for .....I
Simple mechanisms will be supplied for common cases .....I

The conditional control structures are IF...THEN...ELSE...ENDIF and DO CASE...OF...CELSE...ENDCASE (pp. 136-139). They meet the requirements except in relation to discriminated union, which the language does not have.

Once discriminated union is added, providing conditional control structures based on it will be easy.

---

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted). ....P

Termination can occur anywhere in the loop .....I
Multiple terminating predicates are possible .....I
Entry permitted only at the loop head .....I
Simple cases are clear and efficient .....P
Control variable is local to the loop .....F
Control value is efficiently available after termination .....P

SPL/I has fairly clean iterative statements: WHILE, UNTIL, and FOR (pp. 140-146). The control expressions for the FOR are evaluated at the start; better efficiency might result from a restriction against

modifying their values in the loop. The value of the control variable is always available on loop termination, so it is not local to the loop, and this will not be efficient in all cases (e.g., where the user didn't need the value). Another FOR inefficiency is mentioned under J1.

The necessary changes are straightforward, and would probably be fairly easy, but not trivial, to make.

---

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure. ....F

No recursive procedures within recursive procedures .....F
(Maximum depth of recursion can be specified) .....F
(Recursiveness must be specified) .....P

There is no restriction against nesting of recursive procedures (pp. 76-77, 41-44). It is possible to declare a procedure to be NONRECURSIVE, which can greatly increase efficiency.

Adding the restriction would be trivial; changing the compiler to take advantage of the extra information to produce more efficient code would probably be somewhat harder.

---

G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution. ....P

Able to create and terminate parallel processes .....T
Process can gain exclusive use of resources .....T
No parallel routines within recursive routines .....T
No routines within parallel routines .....F
Maximum number of simultaneous instances are declarable .....F
(Access rules are enforced) .....F

Parallel processing facilities are described on pp. 159-196; they appear overly complex for the capability provided, and they clearly depend on an (expensive) heap storage mechanism. A parallel module must be a program declaration (p. 159), which cannot be contained within a

procedure declaration, but can contain another program declaration (pp. 74-75).

Meeting the letter of the requirement would not be hard. Changing the mechanism to be simpler and more efficient would however be a major design task.

---

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program. ....F

Program can get control for any exception .....F
Parameters can be passed .....F
Can get out of any level of a nest of control .....F
Can handle the exception at any level of control .....F

The language has no such facility.

A moderate to major change would be required.

---

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation. ....P-

Priority specification .....F
Synchronization via wait/enable operations .....T
Wait for end of real time interval .....F
Wait for end of simulated time interval .....F
Wait for hardware interrupt .....P
(Can enable and disable interrupts) .....T

There is no direct provision for specification of priorities among parallel paths, although in special situations the counts on signal variables could be used to control priority. The WAIT (p. 195) could in principle be used to wait for a time interval or a hardware interrupt, but there is no language-defined connection between SIGNAL variables and hardware (p. 186). (See the comments on exception handling, requirement G1.)

In this reviewer's opinion, the parallel processing feature of SPL/I should be totally redesigned -- a major task. Just making the modifications explicitly required here would be of moderate difficulty.

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous. ....T

Free format with statement terminator .....T
Mnemonic identifiers possible .....T
Based on conventional forms .....T
Simple grammar .....T
No special case notations .....T
No abbreviations of identifiers or keywords .....T
Unambiguous grammar .....T

SPL/I has a straightforward, mostly clean syntax. For example, IF, WHILE, UNTIL, CASE, and FOR statements must have a corresponding ending bracket (e.g., ENDIF), so that BEGIN...END brackets are unnecessary. For the most part, someone unfamiliar with SPL/I can understand programs in the language.

One unfortunate rule is that a statement may have only one label (p. 47); thus multiple labels must be written L1:; L2:; L3: statement.

---

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences. ....T

There are no such facilities in SPL/I.

---

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset. ....T

---

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals. ....I

Break character exists .....T
(Literals are self-identifying as to type) .....T
(Bit-string literals for any type) .....F

The SPL/I break character for identifiers is the underscore (p. 7). Literals for both primitive and structured types are self-identifying as to type (pp. 105, 111-113).

---

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings. ....I

See page 12.

---

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used. ....I

There are 52 reserved words, which seems reasonable, if not outstanding (p. A-5).

---

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs. ....P+

Uniform comment convention .....T
Look different from code .....T

Bracketed by one or two characters .....I
Can contain any characters .....P
Can appear anywhere reasonable .....I
Terminated by the end of the line .....U
Compatible with automatic reformatting .....I

Comments are enclosed in square brackets; they may contain any characters, except that included square brackets must be nested (p. 32). They can appear between any tokens (p. 30). The manual does not say whether end of line terminates a comment; it seems unlikely.

A trivial change would be required to fully meet this requirement.

---

H8. The language will not permit unmatched parentheses of any kind. ....I

See entire syntax; all brackets must be in pairs.

---

H9. There will be a uniform referent notation. ....I

Fully met. See pp. 116, 124-126.

---

H10. No language defined symbols appearing in the same context will have essentially different meanings. ....I

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program. ....F

SPL/I is full of implementation dependent defaults -- for example, the range and precision of variables (pp. 51-52) and of counters on SIGNAL variables (p. 53), and the association of hardware signals with signal identifiers (p. 186). In addition, the association of a system resource with a resource variable and of a software signal source with a signal identifier are made by programmer conventions (p. 171, 186).

Meeting the requirement would be of at least medium-level difficulty. Not only range declarations, etc., would be required, but also the ranges of arithmetic results would have to be precisely defined in terms of the ranges of their operands.

---

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary. ....P+

Defaults specified for don't care cases .....I
Programmer can override the defaults .....P

Such things as data representation, parameter class (p. 80), storage class (p. 69), reentrancy (p. 76-77), and closed calls are decided by default if left unspecified. Not all can be overridden -- in particular, data representation or closed calls.

Open calls would not be hard to provide, but allowing programmer control over data representation would entail at least a moderate amount of work.

---

I3. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration. ....F

SPL/I has no such facility.

The necessary addition would be fairly easy, if done minimally.

---

I4. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled. ....F

SPL/I has no such facility.

The required addition would not be trivial, but not very hard either.

---

I5. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language. ....F

The whole language is a unit; there is no kernel on which the rest is based.

To define and extract such a kernel would be a major change.

---

I6. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition. ....F

The maximum number of characters in a name is part of the language definition (it is 16 -- see p. 16). Limits on the number of array dimensions, the size of each (p. 65), the number of variables (see, e.g., p. 175), and depth of parentheses nesting in expressions are all implementation dependent.

The additional specification should be easy; the effects on existing compilers might not be.

---

I7. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator. ....I

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs. ....P

No efficiency cost for unused features .....P
Efficient code can be produced for all features .....P

Some inefficiencies in SPL/I are:

1. The direction of a FOR loop is dynamically determined according to the sign of the value of the BY expression; this is unnecessary, and inefficient. Other FOR inefficiencies are mentioned under G4.

2. Pointed-to values can only be in the heap; often more efficient storage discipline would suffice, but SPL/I does not provide for them for these objects (p. 59).

3. A scalar actual parameter can match an array formal parameter (p. 80). We suspect this raises the cost even for the passage of normal arrays as parameters.

4. The rules for GOTO surrogates cause inefficiency.

SPL/I does have a side effect rule which allows efficient code, and allows declaration of non-recursiveness for procedures (both contrary to the Tinman).

At least a moderate effort would be required. The FOR rules would not be hard to change, but providing a choice of storage discipline for pointed-to values, and a good set of GOTO surrogates would be harder (unless the restrictions on GOTO were simply removed, as we believe they should be).

---

J2. Any optimizations performed by the translator will not change the effect of the program. ....U

This is not a language requirement.

---

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions. ....F

SPL/I has no such capability.

The necessary change could be easy to moderately difficult, depending on the degree of integration of SPL/I code and machine code, the protections against loss of optimization, etc.

---

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator. ....F

Encapsulated specification of representation possible .....F
Space/time tradeoff can be specified .....F

SPL/I has no such capability

A moderate to major change is required.

---

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics. ....F

Open/closed properties can be specified .....F
Open and closed versions have the same semantics .....F

SPL/I has no such facility; all procedures are closed.

A moderate change is required.

## Extraneous Features

We recommend that the following features of SPL/I, not required by the Tinman, be kept:

* CFLOAT (complex) type.

* STRING type (assuming that the Tinman requires only a character type, not a character string type).

* SIGNAL, RESOURCE, and PROCESS types. Although not explicitly called for by the Tinman as types, they are needed in the SPL/I realization of other Tinman requirements (parallel processing). They should be redesigned.

* Call by value parameters.

* Bit string literals.

We recommend that the following features of SPL/I, not required by the Tinman, be deleted:

* FRAC (fraction) type. The use of this type is not clear and it can probably be replaced by a fixed-point type.

* DINT (double precision integer) type. The functionality obtained with this type can be assumed by range specifications on integer type.

* CFRAC (complex fraction) and CINT (complex integer) types. These appear to be of extremely limited usefulness.

* Array slices.

It is recommended that the CFLOAT type be retained because it is a numeric type and the user should be able to use the numeric operators with data of that type. If, however, SPL/I is extended to permit extension of built-in operators to user-defined types (which are themselves extensions to SPL/I), then CFLOAT could be deleted from the base language.

## Summary

Despite the fact that it was designed for signal processing applications, SPL/I is a general purpose language, and it is a fairly clean one, with the exception of the parallel processing features. The last may be an unfair criticism, because we have seen no language which provides all necessary mechanisms for parallel processing cleanly and efficiently -- maybe it cannot be done (and therefore should not be a Tinman requirement). Still, we believe it could be done better than in SPL/I.

In many respects SPL/I conforms to the spirit of the Tinman requirements; for example, it has:

* Strong type-checking, without implicit conversions.
* Powerful data structuring tools (without discriminated union or alternate record structures, however).
* Support for parallel processing.
* Structured control mechanisms (although there are some problems with individual mechanisms, as the detailed evaluation shows).
* Variable initialization.
* Attention to separate compilation.
* Recursive procedures, with the ability to declare non-recursiveness.
* Pointer variables, in a way similar to Pascal -- i.e., a clean way.
* Consistent treatment of structures. For example, a function result can be structured, and structured literals can be built.

SPL/I is also a reasonable-sized (not too big), fairly simple, language. It pays for this, however, in missing features. For example, the following are not provided:

* Range and precision specification for variables.
* Powersets, and enumeration types as such.
* Union of types and alternate record structures.
* The ability to define new types, including operations on objects of the type.

* Encapsulated machine code.

* Control over representation of data objects.

* Dynamic upper array bound, or lower bound different from 1.

* Fixed point type.

* Identifiers to stand for constants.

* Conditional compilation.

On balance, SPL/I is a much cleaner, more readable, simpler language, than, for example, CS-4. It even has two very important features that CS-4 does not: Recursive procedures and pointers. It is missing a number of features in CS-4, but most of these are provided in such an awkward way in CS-4 that extending SPL/I would probably result in a cleaner language.

On the other hand, we see little reason to chose SPL/I instead of Pascal as a basis for extension to meet the final version of the Tinman requirements. The chief features in SPL/I which are not in Pascal are parallel processing, structured function results, variable initialization, and multiple storage classes -- and indeed, all but the first are done well. However, these features could easily be add to Pascal. Pascal has, and SPL/I does not, powersets, enumeration types, ranges of values, and alternate record structures. Pascal also has better (more efficient, safer) iteration control structures, and is probably a smaller, simpler language (easier to read and compile) than SPL/I. In addition, Pascal is much better known, and some of the best features of SPL/I have been taken directly from Pascal (e.g., the method of providing pointers).

The SPL/I Language Reference Manual is fairly good -- it is consistently and carefully organized and defines most of the features of the language. However, it could be improved greatly by the addition of more interesting examples (the given examples never provide the answer to a questionable point in the description), the insertion of more summary descriptions (as it is, the reader often has to refer to many different pages to find the answer to a simple question), and an expanded index.

# Comparative Evaluation

# TINMAN vs. FORTRAN

Lloyd W. Campbell
BRL, APG, MD.

August 1976

## 1. Introduction

Section II below is a comparison of the DOD Tinman HOL requirements and the FORTRAN language. The basis of the comparison is primarily the draft proposed standard BSR X3.9 as published in the March 1976 issue of SIGPLAN Notices. The comments provided attempt to indicate how the language meets or does not meet the requirement and the degree of difficulty in changing the language or its implementation to meet the requirement.

Section II Comparative Evaluation

| Requirement | FORTRAN | Comments |
|---|---|---|
| A1. Typed Language | P | All entities have a type fixed at compile time, but explicit typing is not always required. Requiring explicit typing should be a simple change to the language and to a compiler. |
| A2. Data Types | P | Fixed point real data is not provided and its inclusion with automatic scaling is probably a major addition if floating point hardware cannot be used to implement it. Structured data, other than homogeneous arrays and I/O records, are not part of the language except that entities of different type may share elements of an array or even share storage by the use of the EQUIVALENCE statement. I/O records are not treated as a type. Such an inclusion would be a small change to the language and a rather significant change to a compiler. |
| A3. Precision | P | Precision specification has two levels; normal and double. The actual amount of precision provided for either level depends on the processor. Each subprogram may specify its own precision requirements. Permitting more exact precision specification would be fairly simple if double precision is not exceeded. |
| A4. Fixed Point Numbers | F | Fixed point real numbers are not provided. (See A2) |

| | | |
|---|---|---|
| A5. Character Data | F | A partial specification of collating sequence is provided, but it is unalterable. For most processors, permitting a user-defined collating sequence would be rather simple to do. |
| A6. Arrays | P | Except for dummy argument arrays, both the lower bound and the upper bound must be constant and are specified by the user. Dummy argument arrays may have both lower and upper bound specified at entry. Permitting only the upper bound of dummy arguments to vary would be a simple change to the language and the compilers. Permitting the upper bound of all arrays to vary means dynamic storage allocation and that is a more significant change to the compilers. |
| A7. Records | P | The EQUIVALENCE statement permits the mixing of some types within an array. The mixing of character and noncharacter data is not allowed in Fortran. |
| B1. Assignment and Reference | T | Assignment and reference of all predefined types are permitted. |
| B2. Equivalence | P | Fortran does not provide for comparison of logical type entities although some compilers permit it. Permitting such a comparison should be a simple change in the language and in a compiler. |
| B3. Relationals | P | Fortran provides all six relational operators for ordered sets of numbers and for character data. The unordered complex numbers and logical values may not be compared for inequality. |

| | | |
|---|---|---|
| B4. Arithmetic Operations | P | Integer division does not provide a remainder; the division of two integers produces an integer. It should not be very difficult to add such an operation to the language and to a compiler. |
| B5. Truncation and Rounding | T | Fortran does not specify the truncation of the most significant digits. |
| B6. Boolean Operations | P | The "nor" operation is not provided in Fortran, but it would be easy to add. The language permits the "short circuit" mode of evaluation, but does not require it. |
| B7. Array Operations | F | Array operations are not included in the proposed standard Fortran, but such inclusion is feasible and not terribly difficult. A few Fortran compilers exist that have included array operations. |
| B8. Type Conversion | P | Implicit conversions are permitted in assignment statements, but this feature could easily be disabled in a compiler. |
| B9. Changes in Numeric Representation | U | Overflow detection depends upon the hardware. The FORTRAN language includes no provision for detecting such condiditions except on I/O. |
| B10. I/O Operations | T | Dynamic OPEN, CLOSE, and INQUIRE statements along with READ/WRITE statements fulfill this requirement. The ERR= and END= specifiers provide a limited handling of exceptional conditions. |
| B11. Power Set Operations | P | The logical data type with its true and false values is a partial fulfillment of this requirement. |

| | | |
|---|---|---|
| C1. Side Effects | F | Detection of possible side effects is not required and would be a significant addition to a compiler. The FORTRAN language prohibits statements that contain side effects. |
| C2. Operand Structure | T | The interpretation of an expression fulfills this requirement even though the language permits a processor to evaluate any equivalent expression. Parentheses are honored. |
| C3. Expressions Permitted | T | FORTRAN BSR X3.9 X3J3/76 fulfills this requirement. The restriction on subscript expressions mentioned has been removed. |
| C4. Constant Expressions | T | The FORTRAN language does not specify that constant expressions must be evaluated at compile time, but any decent compiler would do so. |
| C5. Consistent Parameter Rules | P | FORTRAN essentially meets this requirement except that alternate return statement labels must be preceded by an asterisk in an argument list. |
| C6. Type Agreement in Parameters | T | FORTRAN requires type agreement and permits passing of subscript limits as arguments or through common blocks. |
| C7. Formal Parameter Kinds | T | FORTRAN has all four classes of "formal parameters" (called "actual arguments" is Fortran). |
| C8. Formal Parameter Specifications | F | Fortran has processor-defined generic functions but does not permit user-defined generic functions. The type of each formal parameter (actual argument) is fixed at compile time. This requirement seems to conflict with C6 which implies that each formal parameter has a type. |

| | | |
|---|---|---|
| C9. Variable Numbers of Parameters | P | FORTRAN permits a variable number of arguments for the max and min functions, but no others. |
| D1. Constant Value Identifiers | T | The PARAMETER statement provides this capability. |
| D2. Numeric Literals | P | Constants of the same form must have the same value within a program, but an input data value of the same form is not required to have the same value as a constant in the program. Adding such a requirement should be easy. |
| D3. Initial Values of Variables | P | FORTRAN has no default initial values and permits user specification of initial values in DATA statements. It does not permit initial values to be specified in the type-declaration statements, but this could be added rather easily and has been added in some compilers. |
| D4. Numeric Range and Step Size | F | FORTRAN has no provision for specifying range or step size except as comments. The degree of difficulty on adding this feature depends on the extent the compiler makes use of this information. |
| D5. Variable Types | F | An array may be a component of an I/O record. Otherwise, FORTRAN doesn't seem to fulfill this requirement. Adding this feature to the language and implementing it is probably a significant task. |
| D6. Pointer Variables | F | It is feasible to add a pointer mechanism to the language and to implement it. It requires a modest effort. |

| | | |
|---|---|---|
| E1. User Definitions Possible | F | Allowing user-defined data types is probably a major addition to most FORTRAN compilers and may not be feasible in some compilers. |
| E2. Consistent Use of Types | F | If user-defined data types are added, they could be added in a manner that fulfills this requirement. |
| E3. No Default Declarations | P | FORTRAN provides a few defaults, but these could easily be removed. |
| E4. Can Extend Existing Operators | F | If user-defined types are added, changes could be made to fulfill this requirement. |
| E5. Type Definitions | F | |
| E6. Data Defining Mechanisms | F | |
| E7. No Free Union or Subset Types | F | |
| E8. Type Initialization | F | |
| F1. Separate Allocation and Access Allowed | F | The FORTRAN language doesn't really distinquish between scope of allocation and scope of access. It only requires that data be available when it is properly referenced. With multi-programming operating systems, the scope of allocation is not fixed nor easily determined and isn't important. This seems to be an undesirable requirement. |
| F2. Limiting Access Scope | T | I think FORTRAN meets this requirement because the scope of a data name is local to a program unit unless it appears in a COMMON statement. The COMMON statement also permits new local names for a nonlocal entity. |
| F3. Compile Time Scope Determination | T | Identifiers have a scope of a program unit or an executable program and this is known at compile time. |

| | | |
|---|---|---|
| **F4.** Libraries Available | T | There exists a large set of FORTRAN application programs and subprograms. |
| **F5.** Library Contents | U | This depends on particular implementations, but it is true for most existing compilers. If it isn't true for some compilers, it is not difficult to add. |
| **F6.** Libraries and Compools Indistinquishable | F | There is no provision for Compools in FORTRAN, but it shouldn't be too difficult to include them. |
| **F7.** Standard Interfaces | P | Fortran provides for interface to several kinds of I/O devices. The general I/O statements could be adapted to special hardware. |
| **G1.** Kinds of Control Strucutres | P | FORTRAN provides for sequential, conditional, and iterative control. It does not permit recrusion and the implementation of recursion is a significant task. It does not provide for parallel processing and has only limited exception handling on normal I/O. |
| **G2.** The Go To | P | FORTRAN has several forms of go to's. It has some of the "undesirables" mentioned, but these could be removed from the language rather easily. |
| **G3.** Conditional Control | P | Conditional control based on the value of a Boolean expression is permitted by the logical IF statement. The computed GOTO statement allows conditional execution of labeled alternatives based on the value of an integer. THE IF-THEN-ELSE structure does not appear in BSR X3.9 X3J3/76, but it has recently (July 1976) been approved by X3J3 for inclusion in the proposed standard. THE ELSE is not required to follo. an IF-THEN but the required ENDIF makes the choice clear. |

| | | |
|---|---|---|
| **G4.** Iterative Control | P | The DO statement provides iterative control. Entry is allowed only at the head of a DO-loop and excessive overhead is not imposed. A loop may be executed zero times or more, but the termination condition is specified at the beginning of the loop. The control variable is not local to the loop and its value is available at termination of the loop. It should not be difficult to make the required changes to the language and the compilers. |
| **G5.** Recursive Routines | F | Recursive procedures are not permitted in Fortran. Adding recursion is feasible. |
| **G6.** Parallel Processing | F | The FORTRAN language assumes sequential execution although there is nothing in the language that prohibits parallel processing. The ability to gain exclusive use of devices normally depends on the operating system. Adding some limited control over parallel processing is feasible if anyone can agree on how it is to be done. |
| **G7.** Exception Handling | P | FORTRAN has a limited exception control on I/O errors and end-of-file detection on input. No mechanism is provided for handling other exception conditions. The difficulty in adding exception control depends largely on the computer hardware involved. |
| **G8.** Synchronization and Real Time | P | FORTRAN has a PAUSE statement that provides delay until operator intervention. A more generalized delay could rather easily be added on computers that have the hardware needed to support this feature. |

| | | |
|---|---|---|
| H1. General Characteristics | P | Although FORTRAN does not have free format and an explicit statement separator, these can be added rather easily and have been adding on a number of computers. The language allows mnemonic identifiers of up to six characters, is easily parsed, does not generally provide unique notation for special cases, and does not permit abbreviations beyond those specified for the statements and keywords. |
| H2. No Syntax Extensions | T | The user cannot change the syntax of FORTRAN. |
| H3. Source Character Set | T | FORTRAN has a limited character set that is well within the ASCII subset. The same character set is used for programming and publication. |
| H4. Identifiers and Liberals | P | FORTRAN provides simple formation rules for identifiers and literals. It doesn't have a true break character although a user may include blank characters in his identifiers; however the computer is not required to retain such blank characters. The inclusion of a break character is a simple change. |
| H5. Lexical Units and Lines | F | FORTRAN permits the continuation of lexical units accross lines, but this could easily be changed. There is no end-of-line character specified in the FORTRAN language, so any provision for its inclusion in a literal string would be processor-dependent. |
| H6. Key Words | P | Key words are not reserved in FORTRAN, but it would be easy to make them reserved. They are informative and never appear in a context where an identifier may appear. |

| | | |
|---|---|---|
| H7. Comment Conventions | P | FORTRAN meets this requirement except it does not include embedded comments and it has two comment conventions. It allows either a C or an asterisk in column 1 to indicate a comment line. It would be easy to add embedded comments and to allow only one comment convention. |
| H8. Unmatched Parentheses | T | |
| H9. Uniform Referent Notation | T | Function references and array element references have the same general form. |
| H10. Consistency of Meaning | T | |
| I1. No Defaults in Program Logic | T | |
| I2. Object Representation Specifications Optional | T | |
| I3. Compile Time Variables | P | The PARAMETER statement fulfills part of this requirement. It may be used to accomplish the objective, but it does not provide predefined meanings for any parameters and it applies only to a program unit, not the whole "program". |
| I4. Conditional Compilation | P | A processor could implement this requirement, but the FORTRAN language makes no such requirement. To be effective, a more general optional compilation scheme is needed. |
| I5. Simple Base Language | F | FORTRAN doesn't seem to have a simple identifiable base and it's not obvious how it could have one. |
| I6. Translator Restrictions | P | FORTRAN has limits on the number of dimensions (7) and the length of identifiers (6) but specifies no other limits and considers them as processor-dependent. However, most compilers have reasonable limits and would probably meet any set of limits set by DOD. |

| | | |
|---|---|---|
| I7. Object Machine Restrictions | T | FORTRAN does not specify limits on things dependent on the object environment. It appears that requirements G7 and G8 depend on the object environment and, according to this I7 requirement, should not be part of the language specification. |
| J1. Efficient Object Code | T | Efficiency has been an important factor in the design of FORTRAN. Inefficient generalities have generally been avoided when they would affect the efficiency of simpler programs. |
| J2. Optimizations Do Not Change Program Effect | T | Any translator not meeting this requirement should be considered as incorrect. |
| J3. Machine Language Insertions | P | The FORTRAN language allows the inclusion of encapsulated functions and subroutines written in other languages. The exact nature of the non-FORTRAN routines are processor-dependent. If machine language is not allowed in same compiler, it should be easy to allow it. |
| J4. Object Representation Specifications | P | FORTRAN doesn't have much in the way of composite data structures, but you can specify the order and width of fields in I/O records and internal files. Any more detailed object specification is not part of the language and is likely to be processor-dependent. |
| J5. Open and Closed Routine Calls | F | A FORTRAN user cannot specify whether a routine is to be coded as open or closed. This would not be difficult to allow on some computers. |
| A1. Operating System Not Required | T | FORTRAN does not require an operating system. |

| | | |
|---|---|---|
| A2. Program Assembly | T | FORTRAN allows independent compilation of program units. |
| A3. Software Development Tools | P | Almost all computers using FORTRAN have some related programming tools available even though proposed language standard does not mention such tools. |
| A4. Translator Options | P | Although the specification of such support software is outside of the province of a language standard, most FORTRAN compilers provide at least some of the required software support. |
| A5. Assertions and Other Optional Specifications | F | Another type of comment would be easy to add to FORTRAN. |
| B1. No Superset Implementations | F | The proposed FORTRAN standard is permissive and permits processor extensions. However, it is usually easy to delete any extensions that are not part of the DOD language. |
| B2. No Subset Implementations | F | FORTRAN does not have a base language and it has a subset level defined for use on small computers. |
| B3. Low-Cost Translation | T | Efficiency of translation has been a factor in the design of the FORTRAN language. In general, translation time depends on the degree of optimization. Although it is not part of the standard, many FORTRAN compilers permit some user-selection on the degree of optimization. |
| B4. Many Object Machines | U | The number of machines that a translator can translate for is independent of the FORTRAN language. |
| B5. Self-Hosting Not Required | U | Most FORTRAN compilers are self-hosting, but this is implemation dependent and not part of the FORTRAN language. |

| | | |
|---|---|---|
| B6. Translator Checking Required | P | It is almost impossible to check for all possible errors, especially when independent compilation of program parts is required as in A2. Most FORTRAN compilers do extensive syntax checking, check type compatibility, and check some of the simpler semantic restrictions. |
| B7. Diagnostic Message | P | The proposed standard suggests a few error situations. Most FORTRAN translators provide a reasonable set of error and warning messages. |
| B8. Translator Internal Structure | T | The proposed FORTRAN standard does not dictate characteristics of translator implementations. |
| B9. Self-Implementable Language | U | The language used to write a translator is implementation-dependent. In general, a true higher-order language is not suitable for writing a translator efficiently as it requires some low-level techniques to do the translation efficiently. |
| C1. Existing Language Features | T | FORTRAN is within the state-of-the-art and is a very practical language. |
| C2. Unambiguous Definition | P | The X3J3 committee has attempted to make the proposed FORTRAN standard unambiguous and clear. The document uses primarily the English language, but also includes a formal definition of the syntax in the form of "railroad" charts. |
| C3. Language Documentation Required | P | The proposed standard is not meant to be a general user manual. However, it includes some tutorial information in Appendices, a simple metalanguage, and includes some examples. |

| | | |
|---|---|---|
| C4. Configuration Management | U | The management of the language by DOD is outside province of the language. |
| C5. Support Agent Required | U | The maintenance of translators is outside the province of the language. |
| C6. Library Standards and Support Requried | U | Standards and agents for common libraries are outisde the province of the language. |

III. Features in FORTRAN Not Required by Tinman

Because many of the Tinman requirements are general rather than specific, it is difficult to select features from FORTRAN that are not needed to fulfill some aspect of the Tinman requirements. However, there are a few significant items and possibly many insignificant items that are not required by Tinman. Some of the more significant items are:

(1) Arithmetic IF Statement - The arithmetic IF statement has been superseded by the logical IF statement and the IF-THEN-ELSE construct and probably should be omitted from the language.

(2) Statement Functions - Statement functions have been superseded by function subprograms and there is no good reason for keeping them.

(3) Block Data Subprograms - The initialization of entities in common should not require the use of a special subprogram and therefore the block data subprogram should be omitted from the language.

(4) ENDFILE Statement - THE ENDFILE statement has no real purpose and should be omitted from the language.

(5) ASSIGN and assigned GOTO Statements - Tinman doesn't appear to require these control concepts, but I recommend that they be retained. In particular, the ASSIGN statement is an easy way to change a print format.

(6) ENTRY Statement - Tinman doesn't seem to require that a procedure can have more than one entry point, but I recommend that the ENTRY statement be retained.

My recommendation of deletion of items (1)-(4) above assumes there is no need for compatibility with existing FORTRAN programs.

## IV. Discussion and Summary

There is some basic difficulty in comparing a language, which is really an abstract entity, with some of the Tinman requirements that refer to implementations of a language. In particular, the requirements on the translator are usually independent of the language involved. It should also be recognized that it is possible that I, and probably some of the other evaluators, have misunderstood some of the requirements. It was rather difficult to relate many of the general and somewhat vague requirements to a specific language.

In summary, my evaluation shows the following:

T for 27 requirements
P for 38 requirements
F for 25 requirements
U for 8 requirements

The most significant changes needed for FORTRAN to meet the Tinman requirements are:

(1) User-defined data types
(2) Parallel processing
(3) Recursive procedures
(4) Automatic scaling of fixed point numbers
(5) Array arithmetic

Although most of the changes needed are minor, the totality of all the changes would require a major effort. However, it should be significantly less than programming a whole new translator.

LTR

## I. Introduction.

The French language LTR is an ALGOL-based language with additional datatype definition facilities and with a rich selection of process management facilities including the concepts of tasks, events, resources, delay, interrupts, and real-time I/O. The process and realtime I/O facilities are the most interesting aspects of the language.

LTR is administered by the French government; for a system to be called LTR it must be certified by the Service Central des Telecommunications et de l'Informatique.

## II. Description.

### A. Lexical.

Source programs are free format on card-images. Card boundaries are ignored.

Identifiers are limited to a maximum of 8 characters.

Keywords are reserved.

Comments are placed between the keyword COMMENT and the

following semicolon.

A simple substitution macro definition facility is provided:

MACDEF <name> % <string> %

Assembler code may be inserted in an LTR source program. Inserted code begins with the keyword CODE and is terminated by the following semicolon. Everything which appears in between is duplicated without modification by the compiler.

B. Datatypes.

LTR is a typed language, but the EQUIV declaration (similar to EQUIVALENCE in FORTRAN) allows the type-checking mechanism to be defeated by overlaying variables of different types.

The following simple data types are provided:

-INTEGER

in 1, 2, and 4 byte lengths.

-FIXED

in 2 and 4 byte lengths.

-REAL

4-byte floating point.

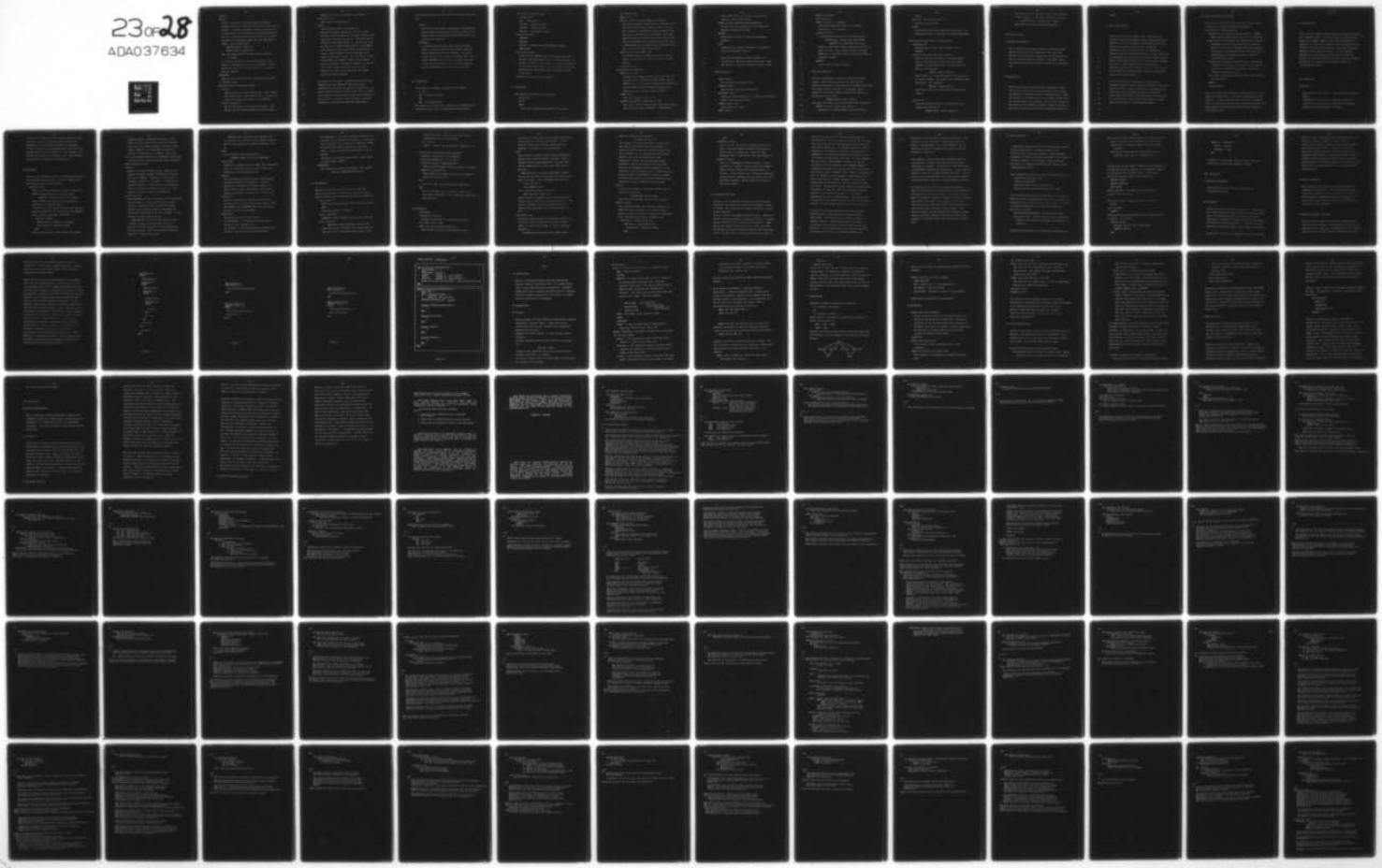
-LOGICAL

bit string, in 1, 2, or 4 byte lengths.

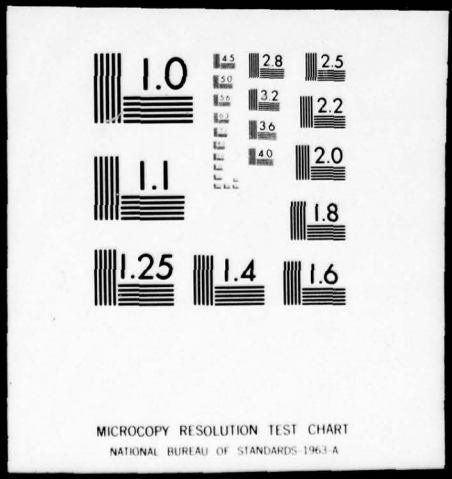
AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

23 OF 28
ADA037634



23 OF 28
ADA037634



-BOOLEAN

-QUALITY

Quality variables correspond roughly to PASCAL enumeration types. There are significant differences, however. A quality variable is always implemented as a byte integer; the encoding for each literal must be given in the declaration; and apparently all quality variables are assumed to range over the same set of values. The declarations:

```
QUALITY(WHITE 1, RED 2) A
QUALITY(BLUE 1, BLACK 2) B
```

make the symbols BLACK and RED synonyms, so that

```
A = BLACK;
```

is a legal assignment, even though BLACK does not appear in the declaration of A, for example. Thus, a significant amount of type information is ignored by the type checker.

-REFERENCE

Both type-restricted and type-unrestricted (REF ANY) variables exist.

Data structuring facilities include:

-arrays

Arrays may be of several dimensions. Upper bounds must be constants; lower bounds are 1. Arrays are stored in column major order, as in FORTRAN.

-simple structures

An LTR structure is a one-level structure. All fields must be simple variables; no array-valued

fields or nesting of structures is allowed.

-structure arrays

These must be 1-dimensional.

-SETS

A SET variable holds a collection of identically-shaped structures. If S is a SET variable, then variables of type REF S point to the individual structures inside S. Structures may be created and destroyed inside S. In this way a SET is similar to a EUCLID COLLECTION or a LIS DOMAIN. LTR, however, provides operations on the SET as a whole. The set appears to the programmer as a doubly-linked linear list. If X is a REF to a SET, then BEFORE X and AFTER X point to the elements which respectively precede and follow X in the SET. It is also possible to iterate over all the elements of a SET, or to search for an element satisfying some condition.

The syntax of structures, structure arrays, and sets requires that the identical structure type be redeclared for each way in which it is used (e.g., as both a simple structure and the object type of a SET). As a result, the compiler must treat all structures of the same shape as being of the same type, ignoring potentially valuable type-checking information.

The following datatypes are included for process-control use:

-EVENT

An event has two states, occurred or not occurred. When an event changes from the not occurred to the occurred state, all processes currently waiting on the event are enabled.

-RESOURCE

A resource also has two states, busy and free. However, there is also a fixed maximum number of users associated with each resource. The total number of units of a resource is conserved: when a process RELEASEs one unit of a resource, only one of the waiting processes (i.e., of the processes which have RESERVEd the resource) is enabled.

C. Expressions.

The following arithmetic operators are provided:

+ - * /

DIV (integer division)

MOD

EXP (exponentiation)

Mixed-mode expressions and assignment are permitted, but the mode-coercion rules are implementation-dependent.

The logical operators are:

LAND,LOR,XOR

CPL (complement)

SLL,SRL (logical shifts)

SLC,SRC (cyclic shifts)

SLA,SRA (arithmetic shifts)

Boolean operators:

GTR,GEQ,

EQL,NEQ,

LEQ,LSS (comparisons yield Boolean result)

AND,OR,NOT

Event expressions:

Events can be combined into expressions using the operators OR (disJunction) and ',' (conJunction). An integer expression can be used as an event expression; an expression with value n is interpreted as an event which occurs n time units after evaluation of the expression.

Conditional expressions are provided.

D. Statements.

The following statements are provided:

-assignment

-block

-GOTO

It is not permitted to branch out of a block.

-IF...THEN...ELSE

-WHILE...DO

-FOR<var>=<exp1>STEP<exp2>UNTIL<exp3>DO<stmt>

The LTR programmer's manual gives a flow chart for the FOR statement which sussests that (a) <exp2> and <exp3> are recomputed on each iteration, (b) the control variable is not protected from beins modified within the body of the loop, and (c) the test is performed at the bottom of the loop, as in a FORTRAN DO-loop, so that the loop executes once even if <exp1> is sreated than <exp3>.

-CASE <int exp> OF BEGIN...END

No runtime check is performed on the <int exp> value. A value out of ranse results in a wild branch.

-procedure calls

Set management statements:

-NEW <id> IN <set>

creates a new element in the <set>, and sets <id> to point to it. Optional clauses AHEAD <exp> or BEHIND <exp> can be used to specify the position of the new element in the (linearly ordered) set. WITH[<exp list>] supplies initialization.

-ERASE <exp>

deletes an element from a set.

-SEARCH <id> GIVING <condition> IN <set>

The <id> is set to point to the first item in <set> which satisfies the <condition>. The optional

clause FROM <exp> can be used to control the starting point of the search.

-FORALL<id>IN<set>WHILE<condition>DO<stmt>

iterates over all elements of a set until the optional <condition> is met.

-RETURN

terminates the block in which the statement appears.

-EXIT

terminates the article (procedure or process) in which the statement appears.

Note that the RETURN and EXIT keywords are interchanged from their conventional uses. Both the documents I have on LTR agree on this usage.

Task Management

-SETEV <event>

set an event to occurred state

-RESETEV <event>

set an event to not occurred state.

-ACTIVATE <event>

enable all tasks waiting on an event, but leave the event in not occurred state.

-TESTEV <event> <id>

test state of an event

-WAIT <event>

-RESERVE <resource>

-FREE <resource>

-STATE <resource> ... RESERVE

indivisibly tests the state of a resource, reserving it if it is free.

-process creation

is syntactically similar to a procedure call:

<Processname> <param list> OPEN|CLOSED

PRIORITY <exp> WAITING <event exp>

creates a new process whose scheduling priority is given by <exp>. The creating process continues (OPEN) or is suspended until the created process is completed (CLOSED).

-MONITOR

passes control to the scheduler.

Interrupt handling:

Interrupt management is based on declared interrupt levels. These levels correspond to the priority interrupt structure of the underlying processor, and thus are highly machine-dependent. An internal name is associated with each interrupt level to be used by a declaration of the form:

DEVICE INTRPT <name> <level>

The named interrupts are then managed by the statements:

-ATTACH [ <intrpt name>,<proc name> ]

associates a procedure with a named interrupt

level.

-DETACH [ <intrpt name>,<var> ]

cancel the association.

-ICTRL

allows controlling and simulating interrupts. The operation of this instruction is system- dependent.

LTR provides three forms of input/output:

-formatted I/O.

This facility is very like formatted I/O in FORTRAN.

-physical I/O under supervisor control.

Each I/O unit is described by a control block, which is declared as a simple structure whose contents is system dependent. An I/O request is made by the statement

IOCS [ <block>,<flag> ]

where <block> is a control block and <flag> is set to indicate whether the request was accepted by the supervisor. The statement

WAITIO [ <block>,<flag> ]

waits for the end of a previously initiated I/O request.

-direct I/O

a direct peripheral device is declared in the system data article by

DEVICE DIRECT <name> <address>

This associates an internal <name> with a physical device <address>. The I/O statements are simply

INPUT!OUTPUT [ <devicename>,<datum> ]

Operation is device and system dependent.

## III. Discussion.

### A. Machine Dependence.

LTR is heavily biased toward machines with 8-bit bytes. Details of the interrupt association and direct I/O features are of necessity highly machine-dependent as is any use of machine-code insertions. The EQUIV feature can produce code which is machine-dependent in nonobvious ways.

### B. Efficiency.

LTR appears to be a highly efficient language. Except for process activation and set management, the run-time environment is completely static and FORTRAN-like. While the language is not type-secure (e.g., EQUIV and REF ANY) no run-time checks are perfomed. The designers appear to have opted for the greatest possible runtime efficiency, with the apparent sole exception of recomputing FOR loop

bounds.

C. Real Time Features.

This is clearly LTR's strong point. The ability to create and destroy processes, process communication and scheduling, interrupt handling, and direct I/O are all present with about as high a level of machine independence as one could expect. The facilities designed into the language apparently are not intended to be implemented under existing operating systems, as the manual states,"Most operating systems have been designed and written with real-time facilities reduced or different from the LTR language concepts. As a result, an LTR real-time program can be properly executed only with a special LTR monitor."

My only reservation is that there seems to be an excessive reliance on the notion of priority, and on the assumption that at most one process can be active at a given time. This assumption, clearly violated on multi-processor systems, does not seem to be necessary for programming in LTR; however, the sample programs given in the manual rely on it consistently.

D. System Programming Features.

LTR is not a particularly good language in this respect -- there are too many syntactic and semantic peculiarities. A few examples:

-identifiers are limited to 8 characters in length. What possible justification can there be for this?

-the strange interpretation for QUALITY data that all identifiers with the same encodings are synonyms. Even if one is going to demand that the encodings be specified, one should consider different QUALITY declarations as producing different types for type-checking purposes.

-limitation of structures to one level. Multi-level (tree-like) structures increase the complexity of a compiler only slightly, and needn't affect the runtime environment at all.

-the archaic implementation of FOR loops with the test at the bottom.

-EQUIV

-free pointers

The point is that the desirable properties of LTR -- and these are significant -- are largely independent of its failings. A modest language design effort to eliminate these failings would improve the language enormously, and thus would be preferable to adoption of the language in its present state.

# I. Introduction.

LIS is a modern system implementation language developed at CII - Honeywell Bull. It builds on many of the desirable features of PASCAL, SIMULA, and the SUE implementation language. It has modern, powerful control structures and a flexible type definition mechanism. The most interesting features of the language, however, are the elaborate mechanism for separate compilation with type-checking between the separately-compiled modules, and the separation of machine- or representation- dependent code into "implementation parts."

# II. Description.

## A. Lexical.

Source text is free format. Line boundaries terminate tokens.

Identifiers can be up to 64 characters long; all characters are significant. Keywords are reserved.

Comments are placed between '$' symbols.

No compile time macro facility exists, though to some extent this feature can be replaced by declaring identifiers to be STATIC (compile-time) CONSTANTS. Program listings can be automatically "pretty printed." In-line machine code insertions are not allowed; use of machine code requires a procedure call. See, however, the material on interfaces in Section II F.

## B. Datatypes.

There are many datatypes in LIS, and a flexible facility for programmer-defined types. A list of these, with brief descriptions of each, follows.

-Discrete types

These are user-defined types which correspond to PASCAL "enumeration types." For example:

```
TYPE day = (Monday|Tuesday|...|Sunday)
```

Unlike PASCAL, no ordering exists among the constants of a discrete type.

In addition to this simple facility, one can give names to subsets of the identifiers of a symbolic type, forming a tree-like structure:

```
TYPE day = UNION (
  TYPE weekday = (Monday|...|Friday);
  TYPE weekend = (Saturday|Sunday)
END
```

Weekday and weekend in this context are called

symbolic ranges. Syntactically they are type names, and they may be used as types in certain contexts such as CASE statements and array index types. However, it is not legal to declare a variable as belonging to a symbolic range type.

-the types INTEGER, BOOLEAN, and CHARACTER are treated as predefined discrete types with special semantics.

-SET OF <discrete type>

-records

Records are called PLEXes in LIS. They are very similar to PASCAL records, and allow for variant parts as in PASCAL. A field of a plex may be a procedure as well as a variable. A field can be declared PRIVATE, in which case it is accessible only to procedure components of the plex. This feature is useful for data encapsulation.

-arrays and rows

Array elements can be of any type, and array bounds can be dynamic. A row is analogous to a dope vector; it specifies (some portion of) another existing array which can then be referenced by subscripted references to the row variable. A row variable can also be declared as

ROW CONSTANT <index type> OF <element type>

in which case the array locations which it specifies may not be changed (if referenced thru the row variable. They may of course be changed directly). Contrast this with

CONSTANT ROW <index type> OF <element type>

in which the row variable itself cannot be changed, but the array locations referenced through it can be.

-TEXT

is a predefined type equivalent to

CONSTANT ARRAY (0..n-1) OF CHARACTER

-index types

Declaring a variable to be of index type associates it with a particular array in such a way as to simplify its use in subscripting.

-domains

are conceptually infinite areas in which dynamic storage allocation is performed. A plex is allocated or freed in a particular domain. A domain differs from a PL/I AREA in that a given domain can contain plexes of only a single type.

-references

Pointer variables are restrited to point to a particular type or into a particular domain. REF, REF CONSTANT, and CONSTANT REF types can all be defined, in analogy with ROWS.

-subprograms

Subprograms may be of three types:

(1) PROCEDURES -- procedure entry does dynamic allocation on a runtime stack.

(2) ACTIONS -- have their workspace allocated in the work space of the enclosing procedure.

(3) COROUTINES -- All the coroutines declared in a procedure form a family of coroutines. A coroutine may be called by the enclosing procedure or resumed by a call from another coroutine in the same family. Storage allocation is the same as for an ACTION.

Parameters are passed by value (IN), result (OUT), or value-result (INOUT).

-CONNECTORS

are restricted procedure variables. For example:

circfcn: CONNECTOR(SIN!COS!TAN)

C. Expressions.

Because of the large number of types in LIS, the expression structure is quite rich. Special syntax exists to denote set and record constants, as follows:

-set primaries:

SET(<type>), denoting the set of all values of the type.

SET(<exp1>!<exp2>!...!<expn>)

-array primaries:

A:ARRAY(1..10)OF INTEGER==(<1..5>==0,<6..10>==1)

-plex primaries:

TODAY:DATE==(DAY==11,MONTH==11,YEAR==76)

Note that several different plex types might have the same set of field identifiers; thus a plex

primary may occur only where its type can be determined from its left context.

-array slices:

A(5..8) denotes the 5th thru 8th elements of A.

A large number of operators are available:

-arithmetic operators: +,-,*,/,MODULO

-relational operators: <,<=,=,/=,>=,>

-logical operators: AND,OR,XOR,NOT

These can be applied to Boolean values or to sets.

-membership relation: IN

tests whether an element value is in a (symbolic range) type or in a set.

REF

If X is a plex, REF X returns a pointer to X.

ROW

If A is an array and T a symbolic range of its index type, then ROW A(T) is a ROW which designates the slice A(T).

D. Statements.

-assignment

-connection statement

assigns a procedure to a connector variable.

-IF...THEN...ELSE...END

-CASE <exp> IN <discrete range> OF ...

The selector expression must belong to some

discrete type; each value of the selector type must correspond to at most one alternative. If the selector expression is out of range, the case statement is treated as a null statement.

-loops

LIS provides a flexible set of looping constructs. These include standard WHILE- and UNTIL- loops, a labeled loop-exit construction for exiting from several nested loops at once, and a FOR- loop of the form

FOR<controlvar>=<iterationrange>LOOP...REPEAT

The control variable is read-only and local to the body of the loop. The <iterationrange> can be

<exp1> TO <exp2>

<exp1 DOWNTO <exp2>

for an integer control variable, or

EACH <discrete range type>

which results in iteration over all values of the discrete type. Note that the EACH construct is not needed in PASCAL, where enumeration types are ordered, but is essential in LIS, where they are not.

-subprogram calls

Parameter correspondence is by keyword in LIS, and parameters can be passed by value (IN), result (OUT), or value-result (INOUT). Thus a procedure declared as

P:ACTION(A:IN REAL,B:OUT REAL,C:INOUT REAL)

might be called by the statement

P( C:=:Z, A:=X, B=:Y )

The operator := copies the actual parameter into the formal at procedure call; the operator =: copies the formal into the actual on return from the procedure; and :=: performs both operations. Default values can be supplied for value parameters, so that the corresponding actual parameters need not be supplied at every call. LIS does not implement call-by-reference. Array parameters are not allowed at all -- but passing an array by value is prohibitively expensive anyway. The effect of passing an array by reference can be achieved by passing a ROW variable by value.

-return

A variation analogous to the labeled loop exit is provided in the form

RETURN FROM <program name>

which allows multilevel returns from procedures.

-WITH statement

This statement allows the temporary binding of an identifier to a variable. It is useful when several accesses are made to fields of a particular plex. WITH statements can be nested, e.g.,

```
WITH dad == kinsman.father DO
    WITH mom == kinsman.mother DO
      IF mom.birth < dad.birth THEN...
    END
```

END

-ALLOCATE and FREE

If P is a plex type, D is a domain of P, and X is REF D, then the statement ALLOCATE X allocates a plex of type P in domain D. Similarly, the statement FREE X releases the plex referenced by X.

-symbolic output

This is a rudimentary printout facility. For a variable of a symbolic discrete type, the symbolic value (and not the internal representation) is printed. Array elements are printed with subscript identification, somewhat like the output of a PL/I PUT DATA statement. Plexes cannot be printed in this way, however.

E. Program Organization.

LIS has a very sophisticated facility for the separate compilation of procedures or groups of procedures within a program. Our discussion of program structure is based on this facility.

First we provide some ALGOL-based motivation. Suppose we wish to compile separately a procedure P4 which is nested inside procedures P1, P2, and P3 as in Figure 1. ALGOL scope rules would say that since P4 is inner to P1-P3, all variables declared in these procedures are accessible to P4. For the sake of separate compilation (as well as

general considerations of program readibility and modifiability) it is desirable to separate thos variables of P1-P3 which should be inherited by inner blocks fro those which should not. (In EUCLID terminology those variables which should be inherited are declared as PERVASIVE). In LIS, declarations of variables of these two classes are physically separated. For the procedure P1, one writes a DATA SEGMENT in which "pervasive" variables of P1 are declared, and a PROGRAM SEGMENT containing the body of P1 and including the declarations of "private" local variables of P1. This is indicated in Fig. 2. Within a program segment, normal scope rules hold except for the "unique visibility rule," which states that an identifier may not be redeclared within its scope. The data and program segments may be compiled separately, if desired. Note the mention of the data segment P1 in the USE DATA context specification associated with P1's program segment. The declarations of separately compilable "inner" procedures P2 and P3 are carried out by repeating this process, as in Figure 3.

A similar effect, but without the nesting of scopes implied above, is achieved with program and data PARTITIONS. A partition groups together a number of related programs. Declarations made in a program partition are visible only to the programs within the partition. Declarations made in the data partition of the same name represent the interface between the

programs of the partition and the outside world. This provides a hiding facility, allowing the effect of modules (EUCLID,MODULA) or classes (SIMULA). As an example, consider the symbol-table handling routines depicted in Figure 4.

Data segments are not statically allocated; they are allocated on a runtime stack when their associated procedures are called. A partition, on the other hand, is allocated at declaration elaboration -- i.e., when the segment in which it is declared gets allocated. This is the reason the symbol table data in the above example is preserved between calls on the table-handling routines.

The implementation of such a separate compilation facility requires that a symbol table be constructed and retained for every data segment or partition compiled. When a partition or segment is named as part of a context (USE DATA) specification, its symbol table must be retrieved so the proper code can be generated to access the variables and procedures of the segment. This facility allows for type checking of parameters passed between separately compiled procedures, a nontrivial problem in a language with a type structure as rich as LIS.

E. Other Features.

A significant feature of LIS is its "two level approach to data-independence." The basic language is a strongly-typed high-level language in which abstract programs are written which are independent of the mapping of data in memory. This mapping is then specified separately, in isolated sections of the program called "implementation parts."

Data implementation parts can specify such properties as

- -encoding for discrete types
- -length and alignment
- -packed/unpacked for arrays or plexes
- -serial/parallel implementation of plex arrays.

Program implementation parts supply bodies of procedures declared in data implementation parts (e.g., nonstandard implementations of allocate and free functions). Within a program implementation part the usual strong type checking is relaxed:

- -BIT,BYTE,HALF,WORD,DOUBLE may be used as types.
- -<exp> AS <type> defeats type- checking altogether, allowing such "unsafe" operations as arithmetic on pointers.

Interface implementation parts allow the programmer to

specify nonstandard linkage conventions. He can specify

-which registers may be modified by a subprogram

-that certain parameters should appear in certain registers

-the machine code sequences used at the call, at procedure entry, and at procedure exit.

LIS does not provide process scheduling and communicating as standard language features. A "nonstandard" feature is proposed in the language manual. The process definition facility comprises the new types:

```
PROCESS ACTION
PROCESS PROCEDURE
REF PROCESS
```

Actual scheduling operations are assumed to be implementation dependent.

The communication facility consists of the type

```
REGION
```

and a statement

```
RESERVE
```

analogous to the WITH statement, which provides mutual exclusion. For example,

```
BUFFER: REGION
   MESSAGE: ARRAY(1..20) OF CHARACTER;
   MSGSENT: BOOLEAN
END
```

```
RESERVE B == BUFFER DO
   B.MSG := 'HELLO';
   B.MSGSENT := TRUE
END
```

A region of a particular process can be referenced indirectly through a REF PROCESS variable.

## III. Discussion.

### A. Machine Independence.

Machine dependent code in LIS is confined to implementation parts.

### B. Efficiency.

LIS has good and bad aspects in this area. The language is strongly typed; all type checking is done at compile time. The classification of subprograms as either procedures or actions is in keeping with the principle of not paying for unused generality -- invoking an action should be considerably cheaper than invoking a procedure.

On the other hand, it is difficult to see how one could

implement LIS's elaborate separate compilation facility without a similarly elaborate runtime display -- effectively (a chain of) pointers to the stack frame of each segment and partition. Similarly, while the complete machine-independence of the LIS language is desirable, the restriction that machine code can only be executed through subroutine linkage may be expensive. Unfortunately, I do not have access to any benchmark data comparing LIS execution speed to, say, an existing PL/I implementation.

C. Real Time Support.

The standard LIS language has no process control or communication facilities. The nonstandard extension described in Section II F does provide the skeleton of a fairly elegant communication mechanism; however, no facilities at all exist for scheduling of asynchronous tasks or for error handling.

D. System Programming Features.

LIS is clearly a modern, appropriate language for the design of large systems. The LIS control structures are appropriate for structured programming; the very flexible type definition facility and the "implementation part"

mechanism allow for top-down, hierarchical program development. The separate compilation mechanism allows programs to be highly modular while still performing typechecking between modules.

Such type checking is a serious problem in a language with as finely differentiated a type structure as LIS. In a language such as RTL or ALGOL 68 in which all types with the same structure are considered the same, it would be possible to redefine types used in separately compiled modules, using a compool mechanism considerably simpler than the LIS separate compilation facility. In LIS, however, separate but textually identical instances of a type definition produce distinct types. This decision is a necessary consequence of the ability to have procedures as components of plexes: otherwise the compiler would be forced to solve an undecidable problem in checking formal-actual parameter correspondence, or to be much more lenient in its type checking. The usefulness of procedure components in plexes, as part of an encapsulation or data abstraction facility, is clear. SIMULA is an outstanding example. These arguments tend to justify what might otherwise seem an unnecessarily baroque separate compilation facility.

```
PROCEDURE P1:
BEGIN

  <declarations>

 PROCEDURE P2
 BEGIN

   <declarations>

   PROCEDURE P3
   BEGIN


     <declarations>

     PROCEDURE P4;
     BEGIN


       <P4 body>

     END

     <P3 body>

   END

   <P2 body>

 END

 <P1 body>

END
```

Figure 1.

```
USE DATA MAIN;
DATA SEGMENT P1;

  <P1 pervasive declarations>

END;
```

```
USE DATA MAIN, P1;
PROGRAM SEGMENT P1;
BEGIN

  <P1 private declarations>

  <body of P1>

END;
```

Figure 2

```
USE DATA MAIN,P1;
DATA SEGMENT P2;

  <P2 pervasive declarations>

END


USE DATA MAIN,P1,P2;
PROGRAM SEGMENT P2;
BEGIN

  <P2 private declarations>

  <P2 program body>

END;
```

Fisure 3.

```
TABLE_HANDLING : Partition ;
```

```
Data Partition TABLE_HANDLING ;
    Type ELEMENT = ...
    INITIALIZE : Action ;
    INSERT     : Action (E : In ELEMENT) ;
    RETRIEVE   : Action (E : Out ELEMENT) ;
    ¢ Specifications of initialize, insert and retrieve ¢
End ;
```

```
Program Partition TABLE_HANDLING ;
    Type CELL = ...
    MAX : Constant INTEGER == ...
    A   : Array (I..MAX) Of CELL ;
    INTERNAL_HOUSEKEEPING : Action ;

    Program INTERNAL_HOUSEKEEPING ;
        ...
    End ;

    Program INITIALIZE ;
        ...
    End ;

    Program INSERT ;
        ...
    End ;

    Program RETRIEVE ;
        ...
    End ;
End ;
```

Figure 4.

# RTL2

## I. Introduction.

RTL2 is a real-time oriented language developed by Imperial Chemical Industries, Ltd. It is ALGOL based, with a record mode declaration capability, a datatype STACK which allows creation and referencing of processes, and with programs structured in such a way as to allow separate compilation of procedures.

## II. Description.

### A. Lexical.

Source programs are free format; card boundaries (newline characters) terminate tokens. Identifiers can be arbitrarily long, and all characters are significant. Keywords are reserved.

Comments are placed between '%' signs and may appear anywhere.

A simple non-parametrized macro facility is provided, e.g.

```
LET nmax = 100;
```

results in the identifier nmax being replaced by the constant 100 wherever it appears.

A sequence starting with the keyword CODE is interpreted as a machine code sequence.

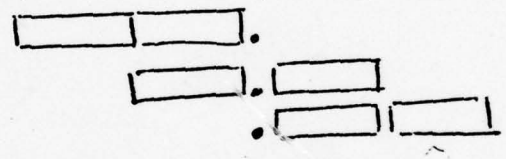
B. Datatypes

Values in an RTL2 program can be of 7 primitive modes:

-REAL (floating point)

-INTEGER

-FRACTION (a value in the range -1<=x<1. Precision is implementation dependent, but a fraction occupies a full word. The double-precision forms BIG and FINE are defined for integers and fractions. BIG adds a word on the left; FINE adds a word on the right. Thus, for example,




-BYTE (an integer in the range 0<=x<256)

-LABEL

-PROCEDURE

-STACK (a value of mode STACK is essentially an activation record stack. See II E).

NORMAL, REFERENCE, and LITERAL variables can be declared for the above modes:

Normal -- The identifier denotes a location which contains a value of the given mode.

Reference -- The identifier denotes a location which contains the address of a location containing a value of the given mode.

Literal -- The identifier denotes a value of the given mode. The literal form is only allowed for labels,

procedures, or stacks. However, a similar effect can be achieved using the parameterless macro facility, e.g. LET N = 10.

The data structure facilities of RTL2 include arrays and records.

Array bounds are constants. The lower bound is implicitly 1. Strictly speaking, only 1-dimensional arrays are permitted. However, array elements may be of any mode except arrays themselves. In particular, they may be records or references to arrays. Thus, a two-dimensional array could be created as

```
ARRAY (10) REF ARRAY REAL b;
ARRAY (10) REAL b1
  .
  .
ARRAY (10) REAL b10
```

Multidimensional arrays are actually implemented in this way, although syntactic 'sugar' allows the more conventional form of declaration

```
ARRAY (10,10) REAL b;
```

Records are treated as mode declarations in RTL2. That is, to declare a record one first declares a record mode, then declares a variable to be of that mode. For example:

```
MODE person ( INTEGER age, ARRAY (8) BYTE name,
   REF PERSON father,mother,
```

```
    BYTE sex)
  person John, Mary;
```

A field of a record can be a simple item or an array of simple items. It cannot be a record or an array of records, although it can of course be a REF to either of these. Note that the restriction that record modes cannot be nested makes the record definition facility of RTL2 somewhat less convenient than that of either PASCAL or PL/I.

## C. Expressions.

Standard arithmetic operators are provided:

+,- (monadic and diadic)

*,/

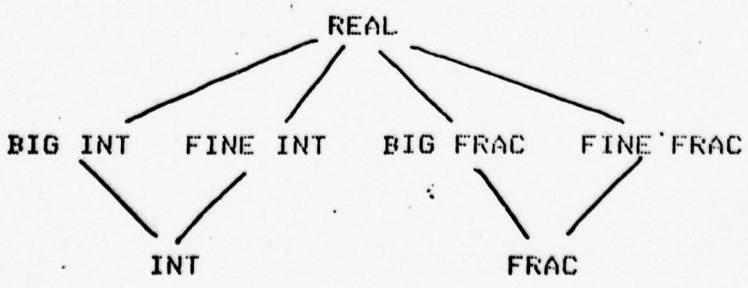
// (integer division)

Only widening is allowed for mixed-mode expressions on normal form values; i.e.

BYTE -> INT -> REAL

FRAC -> REAL

However, conversions between normal and BIG or FINE form are performed via the following (fairly simple) coercion scheme:



Conversion operators are provided among BYTE, INT, FRAC, and REAL.

Logical operators are also provided:

LAND, LOR, NOT

NEV (exclusive or -- "not equivalent")

SLL,SRL,SHL (logical shifts)

SLA,SRA,SHA (arithmetic shifts, 2's complement)

Conditional expressions are also allowed.

D. Statements.

-BLOCK <blockbody> ENDBLOCK

A block defines a scope, and variables declared in a block are reinitialized on each entry. Storage is allocated for all internal blocks of a procedure on procedure entry (this is possible as array bounds must be constants), so there need be no time penalty involved in enterins inner blocks.

-assignment

-GOTO <label expression>

It is legal to exit a procedure using a goto statement.

-SWITCH <expression> OF <label list>

This statement is analogous to a FORTRAN computed goto.

-IF...THEN...ELSE...END

-[FOR <id>:=<exp>[BY<exp>]] TO <exp> DO <blockbody> REP

The loop body is a <block body> and may contain declarations. The control variable is considered local to the loop body.

-WHILE <condition> DO <stmt list> REP

Note the body of a WHILE-loop is a list of statements, and may not contain declarations.

-procedure calls

-RETURN[(<exp>)]

The language does not define statements to perform scheduling or synchronization of tasks; these are assumed to be implementation-dependent supervisor calls. However, stack, SVC proc, and SVC data "bricks" facilitate the definition of such supervisor call routines; see Section E.

E. Program Organization

An RTL2 program comprises one or more separately-compiled modules. Each module consists of a description of its environment (e.g., of its interfaces with other modules) and a number of compilable units called bricks.

Bricks are of three types, as described below:

-A procedure brick is simply a procedure body. Simple variables (but not arrays, records, or procedures)

may be declared inside it, and are inaccessible outside the procedure.

-A data brick is a collection of variable declarations. An array or structure may only be declared in a data brick. Variables declared in a data brick are accessible to all procedures in the same module. A data brick is closely analogous to a named COMMON block in FORTRAN.

-A stack brick is essentially an activation record stack. Whenever a new task is created (RTL2 has no specific instruction for this, but it is assumed that the capability exists as a supervisor call) a stack is supplied as a workspace and a procedure as the code to be executed. Subsequent operations on the task (change priority, terminate, etc.) are then performed by referencing the stack associated with the task.

For communication with the supervisor, one can define SVC procedure and SVC data bricks. A call to an SVC procedure is compiled as a supervisor call. An SVC data brick contains data communicated between supervisor and user programs. Unlike an ordinary data brick, an SVC data brick is private to each task, and is likely to be implemented by mapping it onto a portion of the tasks's work stack.

To combine bricks into a module, one supplies

-environment descriptions, each describing the format

of a brick which is accessed from this module but compiled elsewhere, or describing the format of an available SVC.

-record mode definitions.

-macro definitions

-the bricks themselves.

The above represents a considerable departure from ALGOL scope rules. Blocks within a procedure follow normal scope rules, but all data bricks declared in a module are accessible to all procedures declared in the module, and an arbitrary subset of the externally-compiled bricks can be made accessible by including them in the environment description.

E. Other Features.

Two additional features of RTL2 which deserve to be mentioned are the standard I/O procedures and the standard error recovery system. Note that these both have the status of implementation standards rather than actual language features.

RTL2 has no built in I/O instructions. However, to increase portability, a standard set of I/O procedures is supplied by every implementation. These include routines to read and write fractions, integers, reals and character strings, with input and output files treated as

streams of characters. There are also routines to output spaces and new lines. The interesting fact about these procedures is that they are describable in the source language itself, given a pair of SVC data bricks which perform I/O at the most primitive level, character by character.

There is also a standard error recovery mechanism defined for RTL2. The mechanism is based on the following SVC data brick:

```
SVC DATA RRERR;
   LABEL ERL,
   INTEGER ERN,
   PROC(INTEGER) ERP
ENDDATA;
```

Errors are classified as either recoverable or unrecoverable. On detecting an unrecoverable error, the system assigns an error number to ERN; invokes any monitoring facilities, and then passes control to the label in ERL. A recoverable error results in a call to the procedure in ERP, with the error number as the parameter. ERL and ERP are both normal rather than literal form -- that is, they are label and procedure variables. Thus, a task can create its own error recovery environment by assigning appropriate values to ERN and ERP. It is also a simple matter to save the existing values of ERL and ERP so they may be restored on

exit from the current procedure.

## III. Discussion.

### A. Machine Independence.

RTL2 is not heavily machine-dependent. However, the asymmetry of numerical ranges, where specified, and the arithmetic shift instructions assume 2's complement arithmetic. The range of 0-255 for BYTE values suggests 8-bit bytes.

### B. Efficiency.

The run-time efficiency of RTL2 should be fairly high. The language is strongly-typed; all type-checking (except array bounds and machine-code insertions, of course) can be done at compile time. Memory allocation for data blocks is static, and the language is organized so that entry into inner blocks of a procedure incurs no time penalty. The only dynamic allocation required is for the local variables of procedures. These are allocated on separate stack workspaces for each task. No garbage collection is required.

### C. Real Time Features.

Despite the face that "RTL" stands for "real-time language" there are no explicit process control or interprocess communication primitives in RTL2. However, the STACK datatype provides a very convenient way to identify a task, pass it as an argument, etc. The justification for not defining a set of scheduling and communication primitives is that different operating systems provide different capabilities and features in this area, so that any fixed set of primitive operations is likely to be difficult to implement on at least some existing operating systems. This is probably a reasonable position to take if the language is intended to be used primarily under operating system control. On the other hand, if the intended use of the language is primarily on dedicated processors (e.g., embedded computer systems) then there seems to be no reason not to choose some fixed set of scheduling and communication primitives and make them the language standard.

RTL2 does not provide control over mapping of structures in storage. Thus, for example, it is difficult in the language to overlay a structure on a control register to provide symbolic access to fields and bits within the register. Accessing such fields by explicit sequences of shift and masking operations is undesirable in a high level language, as it adversely affects program readibility and modifiability.

There is also no explicit mechanism for naming interrupts or interrupt classes and associating procedures with them, as would be very convenient in direct

The RTL2 standard error recovery system is reasonable but limited; it probably should not be viewed as a general interrupt or exception handling facility. For certain types of error, a number identifying the type of error is sufficient to enable the error to be processed correctly. In other cases, considerably more information is needed, and the RTL2 error handler simply does not provide it. The need for additional information results in a proliferation of special- purpose pseudo-variables in PL/I (ONCODE, ONSOURCE, ONCHAR, etc.) and an ad hoc set of rules for whether operations are retried, resumed, or aborted on return from the exception handler. In this regard, the design of an exception handling system which is adequate but not baroque is a difficult problem [6]. Note also that RTL2 does not provide an indivisible operation for saving the error-handling environment and creating a new one -- this operation involves two assignments to procedure variables -- The programmer must ensure that an exception cannot occur while the environment is being modified, or that the environment is always in a consistent state able to process an error.

D. System Programming Features.

RTL2 has a number of features which could make it appropriate for use in programming large systems; but it also has several drawbacks. It is a strongly-typed, high-level language. It has adequately structured control statements; however, its type definition facilities are not up to the state of the art, even in terms of what can be implemented very efficiently. The lack of any control over mappings of structures into memory is likely to increase the use of machine code insertions, adversely affecting program modularity and transportability. The separate compilation facility is acceptable provided a library or comool facility -- not mentioned in the manual -- is available for the copying of type and data brick declarations. Finally, the lack of language defined scheduling and communications primitives is inconvenient and is likely to seriously restrict portability.

DRAFT Evaluation of ECL With Respect to the "TINMAN" Requirement for a Common Programming Language for DoD

Each requirement is restated in a form which seems to more clearly exhibit its structure. The words and punctuation have not been changed but its appearance, using indentation, has been changed.

The grading scheme has been extended, where

E means that the requirement may be satisfied by extension

S means that the SPECL* facility will be required

X means that an exception is taken to the requirement

Following the grade is an explanation which refers to specific features which satisfy the requirement, features which are unnecessary, extensions that need to be made and exceptions that are taken. Additional notes are inclosed in angle brackets ("<" and ">").

The evaluation is not complete, but is intended to indicate how each section is to be further elaborated. Although the requirements appear to be well organized, a study of them to the extent necessary to produce this partial evaluation has revealed that they are not well organized. Requirements address several issues and some of the issues are not necessarily related. Explanations of the requirements may contain requirements themselves which apply to the requirement being explained or to other requirements or explanations. The basic terminology being used is not defined. As a result it is not considered useful to complete the elaboration of the evaluation within the current structure of the requirements.

Dave Fisher has informed me that a revised requirement is being prepared which is expected to have a structure more appropriate for evaluation. I expect to be receiving a draft and using it for further work. When the evaluation is complete, each section will specifically address each issue presented in the requirement by identifying the relevant notions in ECL and then giving specifics based on ECL reference material.

STEPHEN L. SQUIRES

*The basic ECL compiler assumes that the complete ECL run-time system is available for compiled code on the PDP-10. The SPECL facility includes a compiler which may produce code for an environment without a run-time system and will have facilities for tailoring highly efficient code to take advantage of features of the machine architecture for certain specialized operations; further, the code generation mechanism will be table driven, permitting relatively inexpensive development of code generators for a variety of machines.

A1.

The language will be typed.

The type (or mode) of all
    variables,
    components of composite data structures,
    expressions,
    operations, and
    parameters
will be
    determinable at compile time and
    unalterable at run time.

The language will require that
    the type of each
        variable and
        component of composite data structures
be explicitly specified in the source programs.

.a T (with generic modes)

.b

Every object in the language has a type (refered to as it's MODE). The built-in routine MD returns the mode of any object.

The system does not require that all MODEs be determinable at compile time but is able to exploit this information when it is available.

For those cases in which a variable is to be bound to a value whose mode is not certain until the program is run, the language provides a special class of modes, called generic modes. Generic modes may only appear as the declared modes of local variables and formal parameters or as the result mode of an explicit procedure. Each is actually a set of modes representing the alternative mode possiblities for the variable or result. Generic modes are created by the built-in function ONEOF.

Once an alternative has been chosen during a variable declaration or on procedure entry, the mode of the variable is fixed for its lifetime. That is its mode is uncertain only before the variable is bound, not afterwards. Thus, there are, in fact, no values of generic mode. Nor may generic modes appear as component modes in the forms STRUCT, VECTOR, SEQ, PTR, or ONEOF.

There is a mode constant of class generic called ANY. When the expected mode is ANY, values of any mode whatever will be passed without conversion or type fault. A variable so declared is subject to the same restrictions that apply to other generic modes: after being given an initial value, the variable's mode is fixed.

The mode of a variable, parameter, expression, or operation may be generic; but the mode of a component of a composite object may not be generic.

The mode of each variable and component may be explicitly specified in the source program.

A2.

The language will provide
data types for
integer,
real (floating point and fixed point),
boolean, and
character, and
as type generators, will provide
arrays (i.e., composite data structures
with indexable components
of homogeneous type) and
records (i.e., composite data structures
with labeled components
of heterogeneous type).

.a E

.b

The language's built-in MODEs include
INT for integer,
REAL for floating point,
BOOL for boolean, and
CHAR for character.

Fixed point may achieved by extension.

The language's built-in classes or MODE-valued forms include
ROW for arrays and,
STRUCT for records.

<note the use of classes when talking about the mode constructors,
this defers the need to introduce SEQ and VECTOR>

A3.

The source language
will require
global (to a scope) specification of
- the precision for floating-point arithmetic and
will permit
the global precision to be overridden by
precision specification for individual variables.

These specification will be interpreted as
the maximum precision required by the program logic and
the minimum precision to be supported by the object code.

.a E, S

.b
The specification of precision of floating-point arithmetic operations and variables may be achieved by extension. The interpretation to be given by the compiler for a paticular machine could be achieved with SPECL.

A4.

Fixed-point numbers
will be treated as exact quantities which have a
range and a
- fractional step size
determined by the user at compile time.

Scale-factor management
will be done by the compiler.

.a E

.b
This requirement could be satisfied by the fixed-point extension.

A5.

Character sets
will be treated as any other enumeration type.

.a E

.b

The treatment of character sets as any other enumeration type could be achieved by using the enumeration extension.

A6.

The language will require
user specification of
the number of array dimensions,
the range of subscript values for each array dimension, and
the type of each array component.

The number of dimensions,
the type, and
the lower subscript bound
will be determinable at compile time.

The upper subscript bound
will be determinable at entry to the array allocation scope.

.a E

.b

The mode-valued forms of class ROW could be used as the basis for defining a mode-valued form having the desired behavior with an operator name such as ARRAY.

A7.

The language will permit
records to have alternate structures,
each of which is fixed at compile time.

The
name and
type
of each record component will be
specified by the user at compile time.

.a T

.b

Alternate structures for records are provided for by defining a generic mode using the built-in function ONEOF; Each of the alternatives corresponds to one of the alternative structures.

The name and mode of each component of a STRUCTure must be sepecified when it is defined.

<need to give consideration to the explaination of this requirement where it refers to (CMS-2 and JOVIAL) OVERLAY and FORTRAN EQUIVALENCE this may be raising the issue of sharing the same site with objects whose modes are not related through ::
this could intertere with representation independence>

B1.

Assignment and reference operations will be
automatically defined for all data types
which do not manage their data storage.

The assignment operation will permit
any value of a given type to be assigned to a
variable,
array or record component of that type or of a
union type containing that type.

References will retrieve the last assigned value.

.a T (with an alterative to union type)

.b

Assignment and reference operations are
automatically defined for all modes.

The assignment operation permits
any value of a given mode to be assigned to a
variable or
proper object of that mode.

Components of compound objects are proper objects.

There are no objects of generic mode.

<The idea of assigment to a union type containing that type
seems to violate one of ECL's object axioms>

<Generic modes indicate the alternative modes that may be used,
once an object is bound using a generic mode, it mode is
one of the alternatives and assignment behaves accordingly.>

Reference to proper objects retreive the last assigned value.

<This does not consider the posibility of using mode behavior functions.>

B2.

The source language will have
a built-in operation which can be used to
compare any two data objects (regardless of type)
- for identity.

.a T

.b
The built-in routine = (or EQUAL) is
TRUE for objects of primitive mode
iff they are identical values
(except for INT and REAL conversions)
TRUE of pointers
iff they point to the same object
TRUE of composite objects
iff they have the same mode and length and
corresponding components.
The INT and REAL conversions could be removed so as not
to conflict with the requirement against implicit conversion.

<there are some implict conversions which do not violate the
intentions of making all conversions explicit,
that is the conversions among modes of class PIK>

B3.

Relational operations
will be automatically defined for
numeric data and
- all types defined by enumeration.

.a E

.b

Relational operations of
GT for greater than,
GE for greater than or equal,
LE for less than or equal, and
LT for less than or equal
are built-in routines for INTs and REALs.

Their definitions may be extended for
extensions to other numeric data and
enumeration modes.

B4.

The built-in arithmetic operations
will include:
addition,
- subtraction,
multiplication,
division (with a real result),
exponentiation,
integer division
(with integer or fixed-point arguments and remainder), and
negation.

.a E

.b
The built-in arithmetic routines
include:
+ for addition,
- for subtraction and negation,
* for multiplication,
/ for division
(if both arguments are INT
then the result is an INT;
if one is REAL
then the result is a REAL)

The exponentiation operation is available from a library
but could be built-in.

These operations could be extended for numeric data extensions
and redefined to have the desired behavior for the operations
of division and interger division.

B5.

Arithmetic and assignment operations
on data which are within the range of specification of the program
will never truncate
- the most significiant digits of a numeric quantity.

Truncation and rounding
will always be
on the least-significant digits and
will never be
implicit for integers and fixed-point numbers.

Implicit rounding
beyond the specified precision
will be allowed for
floating-point numbers.

.a E

.b
Fixed-point numbers are assumed to exist by extension.

The existing implementation uses the operations on
the underlying machine (PDP-10) although
the operation definitions could be made
to satisfy the requirement if they differ.
<some more details might be appropriate>

B6.

The built-in boolean operations
will include
and,
- or,
not, and
xor.

Operations such as and and or on scalars
will be evaluated in short-circuit mode.

.a E

.b
The built-in boolean operations
include
AND for and,
OR for or, and
NOT for not.

An operation XOR for xor could be built-in.

The AND and OR operations are defined over BOOLs and
are evaluated in short-circuit mode.
They may be used either as infix operations or
as a function call with a variable number of operands.

B7.

The source language will permit
scalar operations and
assignment
on conformable arrays and
will permit
data transfers between
records or
arrays
of identical logical structure.

.a E

.b

Scalar operations on conformable arrays can be added.

Assignment is defined for objects of the same mode and size.

Data transfers between records or arrays of identical logical structure may be achieved using the built-in routine REPACK, or if required assignment can be extended.

B8.

There will be
no implicit type conversions, but
no conversions will be required
- when the type of an actual parameter
is a constituent of a union type
which is the formal parameter.

The language will provide
explicit conversion operations among
integer,
fixed-point, and
floating-point data
between
the object representation of numbers and
their representations as characters, and
between
fixed-point scale factors.

.a E

.b

There are no implicit mode conversions for unextended modes except when the following expected modes are bound to the corresponding actual values:

| Expected Mode | Actual Value |
|---|---|
| NONE | any value |
| INT | a REAL |
| REAL | an INT |
| PTR(..., m, ...) | any pointer to an m |
| REF | any pointer |
| 'INT | a VECTOR(n, BOOL) for n LE wordsize |

Of these only the INT and REAL conversions would have to be removed and made explicit to satisfy the requirement.

The conversion to NONE is usually used when a routine has a result mode of NONE because it does not return a value (other than the value NOTHING).

The class of GENERIC modes may only appear as expected modes, no conversion takes place, and the mode of the actual value must agree with one alternative of the generic mode.

Explicit conversion among integer, fixed-point, and floating-point may defined with the numeric extension.

Explicit conversion between characters and intergers is provided by the built-in functions CHAR\INT and INT\CHAR.

Explicit conversions between fixed-point scale factors may be defined with the fixed-point extension.

<Exception might oe taken with respect to the requirments position on implicit conversions.

The apparent reason for avoiding implicit conversions is contained in the requirement explaination that without explicit indication in the program, they are not only error prone but can leao to unexpected run time overheao.

The usual reasons tor wanting implicit conversions include the desire to avoid explicit statement of what might be considered natural or frequent conversions, and the desire to factor the constant from the variable.

This is another apparent delema. One way to resolve it is to recognize that one might like to have the economy of notation while retaining the ability to have it explicated when desired. This facility is contained in the notion of the shadow which is part of the program manipulation system.>

89.

Explicit conversion operations
will not be required between numeric ranges.

There will be a
run time exception condition
when any
integer or
fixed-point value
is truncated.

.a E

.b

The numeric extension could be defined so that explicit conversions between numeric ranges is not required.

The existing implementation recognizes the exception conditions for integer overflow which corresponds to integer truncation.

Fixed-point exception conditions can be provided in their definition.

B10.

The base language will provide
operations allowing programs to interact with
files,
- channels, or
devices,
including terminals.

These operations
will permit
sending and receiving both
data and and control information,
will enable programs to
assign and
reassign I/O devices dynamically,
will provide
user control for exception conditions, and
will not be
installation-dependent.

.a E

.b
The existing facilities for PORTs provides for unifying
the various interactions required as an abstract device.

Given a specification for what is desired it could be added.

<Check news for recent additions for handling channels>.

<The existing I/O facilities show the potential for satisfying
these requirements, and are orthogonal to the base language
when viewed through the mode facility>

<The explaination of the requirment is satisfied by
the existing facilities.
There are a small number of generic I\O operations
with the appropriate device and exception parameters.
More specifically

The built-in mode PORT defines a data object
used to associate the user's program with a source of
input data or sink for output data. The source or sink
may be terminal device (for instance, the user's
terminal, the printer, or a connection established to
another computer system), a file on an external storage
device, or a temporary file held in the user's main
storage.

In order to establish a PORT, the routine OPEN is
normally used. At the termination of input-output,
the CLOSE routine is normally used. The DRAIN
routine may be used to force any output data held
in core buffers out to the device without closing the file.
This procedure is often required in interactive use
of a terminal device.

The MAKEPF routine allows main storage to be used for a temporary file.

Unless the user so directs, reaching end of file will result in a system error in the input-output routine being used. The user may, however, supply a form to be evaluated at the end of file. This form may either be associated with the PORT or with some scope in which the PORT is being accessed.

The facility provides for SYMBOLIC, BINARY or byte access, and may be used for direct access.

INCHAR and OUTCHAR

INOBJ and OUTOBJ

SET\BYTE

...more can be said, the intent of this is to show the set of generic opeations>
<another point to be made relates to the explaination of the requirement ...

In conjunction with requirement E1,
this requirment permits user definition of unique equipment and its associated I/O operations as data types within the syntactic and semantic framwork provided by the generic operations.

Is precisely the point of view taken by ECL >

B11.

The language will provide
operations on data types defined as
power sets of enumeration types (see E6).

These operations will include
union,
intersection,
differences,
complement, and an
element predicate.

.a E

.b
Enumeration types could be added by extension and defined
to have the required behavior.

C1.

Side effects
which are dependent on the evaluation order
among the arguments of an expression
will be evaluated left-to-right.

.a T (except for the explicit use of EVAL on UNEVALed parameters)

.b

Arguments of an expression correspond to the arguments of
the operators of the expression, and operators are
defined as routines.
The order of arguments to an operator is the same as
the order of arguments to the routine that defines it.
The arguments of a routine are evaluated left-to-right
providing they are bound evaluated. All of the operations
required have their arguments evaluated left-to-right.

The binding of a parameter as UNEVALuated
provides for (among others) the EVALuation of the actual
parameter whenever desired.

C2.

Which parts of an expression
constitute the operands to each operation within that expression
should be obvious to the reader.

There will be few levels of operator hierarchy and
they will be widely recognized.

.a E

.b

The language provides for the usual operations, levels of hierarchy,
and parenthesis. Any specific hierarchy required is easily provided.

The facilities for establishing the fixities of operators could
be extended to restrict their usage as indicated in the requirements
explaination.

<The requirement itself deserves a T response, but the further
restrictions in its explaination with regard to redefining
new operator precedence rules or precedence of existing operators
forces an E response.

It may be appropriate to take exception, X, to this requirement
explaination and exhibit the delema that motivates it with a
solution provided by the shadow.>

C3.

Expressions of a given type
will be permitted anywhere in source programs
where both
-constants and
references to variables
of that type are allowed.

.a I

.b

A form in the language is a syntactically complete unit, and each form represents a value. Forms may be combined according to the composition rules of the language to obtain larger forms. The notions of expressions, constants and references to variables are contained in the notion of forms.
Forms may be used anywhere in the language were a value is required. As a result forms which evaluate to a given mode may be used where a value of that mode is expected.

C4.

Constant expressions
will be allowed in programs
wherever constants are allowed, and
constant expressions
will be evaluated before run time.

.a T

.b

Constant expressions are included in the notion of expressions and are therefore allowed wherever constants are allowed.

The compiler may be directed to evaluate constant expressions.

<using the UNEVAL binding it is possible to semantically restrict what kind of expressions are allowed for a particular parameter>.

C5.

There will be a consistent set of rules
applicable to all parameters, whether they be for
procedures,
- types,
exception handling,
parallel processes,
declarations, or
built-in operations.

There will be no special operations
(e.g., array substructuring)
applicable only to parameters.

.a T

.b

There is a consistent set of rules for the generation of all objects.
The notion of object generation applies to parameters of ROUTINES
and DECLarations.
MODE ROUTINEs are ROUTINEs that return MODEs.
Exception handling is defined by a FORM.
Parallel processes are defined as ROUTINES.
Built-in operations are defined as ROUTINES.

There are no special operations applicable only to parameters.

<In fact everyting can be thougnt of as parameters to something.>
<It is not clear what is meant by exception handling parameters
in this requirement, however the generality trap facility would
lead to the expectation that it could satisfy requirements for
exception handling.>

C6.

Formal and actual parameters
will always agree in type.

The number of dimensions for array parameters
will be determinable at compile time.

The size and subscript range for array parameters
need not be determinable at compile time, but
can be passed as part of the parameter.

.a E

.b

Formal and actual parameters must always agree in type
for unextended modes, except for the conversions
between INTs and REALs which could be removed.

If the mode of the formal parameter of a routine
has a fixed number of dimensions then that number will be
determinable when the routine is compiled.

The array extension may be defined so that its size and
subscript range need not be determinable when compiled,
but can be determined from the actual parameter.

<The only reason that an E is given here is because of reference
to arrays which are an extension, based on the mode class ROW,
to provide for subscript ranges.>

C7.

There will be only four classes of formal parameters.

For data,
there will be
those which act as constants,
representing the actual parameter value
at the time of call, and
those which rename the actual parameter,
which must be a variable.

In addition,
there will be a
formal parameter class for specifying the control action
when exception conditions occur and
there will be a
class for procedure parameters.

.a E

.b

The bindclass BYVAL provides the capability to act as a constant in the sence that assignments made within the routine to the to the formal parameter do not produce a side effect, but do change the value of the parameter within the body of the routine. The constant behavior within the body could be added. The initial value of the formal parameter is the actual parameter vaue at the time of call.

The bindclass SHARED may be used to rename the actual parameter which need not be a variable. If the acutal parameter evaluates to a proper object then assignments to the formal parameter are the same as assignments to the actual parameter.

Exception handling is provided for through the use of disginguished identifiers which are bound to forms to be evaluated when the exception occurs. As a result no special parameter class is required for specifying the control action.

Actual parameters may be of any mode including ROUTINE and PROCs. The PROC modes may be used to specify the parameter structure.

<The only reason for E is the requirement for constant behavior for other than manifest constants.>

C8.

Specification of the
type,
range,
-precision,
dimension,
scale, and
format
of parameters will be
optional on the formal side
(i.e., in the procedure declaration).

None of them will be alterable at run time.

.a E

.b

Assuming that these notions have been defined
they may be specified as desired on the formal side
in the mode part of the formal parameter specification.

Their behavior may be defined to be invariant once a
binding is made.

C9.

There will be provision for
variable numbers of arguments,
but in such cases
all-but a constant number of them must be of the same type.

Whether a routine can have a variable number of arguments
must be determinable from its description, and
the number of arguments for any call
will be determinable at compile time.

.a E

.b
There is provision for a variable number of arguments
using the formal parameter bindclass LISTED
as the last parameter.

The existing facility allows any mode of
the actual parameter to be used since the
list of parameters is left unevaluated.

The requirement could be satisfied by allowing
a specific mode to be specified and having the
variable parameters evaluated and placed in
a SEQuence.

The fact that a routine can have a variable number of arguments
may be determined from its formal parameter specification.

The number of arguments for any call can be determined from
the inspection of the call.
<Generic modes could be excluded, or required to be homogeneous,
or a heterogenerous structure allowed>

D1.

The user will have the ability to
associate constant values of any type with identifiers.

.a E

.b

An extension could be defined which would allow a proper object to have the behavior of a pure value. Any object can be associated with an identifier.

The compiler can be directed to exploit this information.

<Some details of how this might be done could be given.>

D2.

The language will provide
a syntax and
a consistent interpretation
for constants of built-in data types.

Numeric constants will have
the same value (within the specified precision)
in both
programs and
data (input and output).

.a I

.b

The language provides a syntax and a consistent interpretation
for constants of built-in modes. These include:

booleans: logical truth values
TRUE and FALSE

integers
sequences of digits

reals
sequences of digits including one radix point and
possibly with scientific notation

characters
a percent sign (%) followed by any character

references: pointers to data objects
NIL is the only reference constant and
it represents a pointer to the empty value

the empty value
NOTHING

modes: values which describe modes
Mode constants represent the built-in modes:
BOOL, INT, REAL, CHAR, REF, MODE, PROC,
SYMBOL, STRING, ROUTINE.
One mode constant, ANY, plays a special role
in the language, though there are not values
with this mode.

symbols: which permit efficient representation of
symbolic expressions
Sequences of characters enclosed in
double quotation marks (")
with % as an escape to include
the next character (for %" and %%).

strings: arrays of characters
Sequences of characters enclosed in
single quotation marks (')
with % as an escape to include
the next character (for %' and %%).

**procedures:** values which specify a transformation from a set of input values to a result.

Procedure constants are procedure definitions. They are not elementary constants since they are composed of other forms.

D3.

The language will permit
the user to specify the initial values of individual variables
as part of their declaration.

Such variables will be
initialized at the time of their apparent allocation
(i.e., at entry to allocation scope).

There will be no default initial values.

.a T, X

.b
The language permits
the user to specify the initial values of individual variables
as part of their declaration as part of the generation
specification of the declaration.
<more could be said about the dinoclasses: ...>

All variables are initialized at the time that they are generated.

Default initial values are provided for all modes.
Exception is taken to the explaination of the requirement
that there

D4.

The source language will require it users
to specify individually
- the range of all numeric variables and
- the step size for fixed-point variables.

The range specification will be interpreted as
the maximum range of values
which will be assigned to variables
the minimal range
which must be supported by the object code.

Range and step-size specifications
will not be interpreted as defining new types.

.a E (part of a specific extension)
.D

These requirements may be satisfied by the extension
for numeric and fixed-point variables.

D5.

The range of values
which can be associated with a
variable,
- array, or
record
component
will be
any built-in type,
any defined type, or
a contiguous subsequence of any enumeration type.

.a T (assuming that enumeration types are defined)
.b
Assuming that enumeration modes are defined
this requirment is satisfied by the fact that
built-in modes, defined modes, and extended modes
are all modes and objects.
A variable can be associated with an object of any mode
(including mode, since mode is a mode).
A component of an array or record may be any nongeneic mode.

D6.

The language will provide
a pointer mechanism
which can be used to build data with
- shared and/or
recursive
substructure.

The pointer property
will only affect the use of variables
(including array and record components)
of some data types.

Pointer variables will be
as safe in their use
as are any other variables.

.a T

.b

The language provides a pointer mechanism which can be used to build data with shared and/or recursive substructure.

The mode constant REF denotes a primitive mode which can be intuitively described as the set of objects which can point to arbitrary objects.

The built-in pointer-generator ALLOC creates a new object of specified mode and initial value in the heap and returns a pointer to the object.

The destination object of any pointer resides in the heap, and the value of such an object remains accessible so long as some pointer-valued variable exists, pointing to it.

The link between a pointer P and an object Q to which it points is provided by a built-in function VAL, i.e., VAL(P) = Q.

There are other pointers which are more restricted than REFs in the mode of the objects they can reference.

The built-in operator PTR takes a mode m as argument and produces the mode: the set of pointers restricted to point to objects of mode m.

The operator PTR can be given more than one argument and when so invoked it produces a mode defining pointers which may reference objects of any of the specified modes. The mode of such a pointer is said to be a united pointer. The particular REF returned by the ALLOC operator will be convertible to the mode PTR(m) which is convertible to a united pointer which contains m.

E1.

The user of the language
will be able to define
new data types and
operations
within programs.

.a T

.b

The user of the language is able to define new modes and operations within programs.

The term mode is used in the language to designate a certain class of objects: objects corresponding to the intuitive notion of data type.

Variables can be declared of mode MODE and restricted to MODE values.

The primitive modes are those denoted by the mode-valued constants INT, REAL, BOOL, CHAR, NONE, REF, and ANY.

New modes may be produced using the built-in mode-constructing operators VECTOR, SEQ, STRUCT, PTR, UNION, and PROC.

From these other mode-valued forms can be synthesized using conditional expressions, functional compositions, and recursion.

<some details of the built-in mode-producing operators>
<some details of the mode behavior facility>
<some reference to the extension to provide for arbitrary operator definition>

The term operator is used in the language to designate an identifier for a routine which may have parsing properties such as prefix and infix.
Such operations have the appearance and costs of built-in operators.

A particular identifier can be given the parsing properties using the operator-defining routines
NOFIX, PREFIX, INFIX, MATCHFIX, SUFFIX.

<some details of the built-in operator-defining routines>
<possibly mention PLUSFIX>
<further justify the same appearance and costs
all operators derive their parsing behavior from the fixity provided by the operator-defining routines, and
all operators have the behavior of a ROUTINE
basic work has been done which shows how the data abstraction mechanisms and access control mechanisms which have been appearing recently in the literature may be realized in terms of the existing mode facility of the language.>

E2.

The use of defined types
will be indistinguishable from built-in types.

.a T

.b
The use of defined modes is indistinguishable from
built-in modes.

All modes have the mode of MODE.
<the following is a restatment of some of the requriements
explication of this requirement to emphasize that
it is reasonable to achieve the expectations discussed>

The built-in features of the language behave as if
they were defined using the facilities available
for user-defined modes and operations.

Both appear to be treated in the same way
throughout the language, so that the base language,
standard application libraries, and application programs
are treated in a uniform manner by the user and
evaluators of the language.

The mode-behavior facility provides encapsulation capabilities
and the means to specify special functions such as
generation, selection, conversion, assignment, printing, dimensions.

The language seems to have the potential to contain all the
esential power desired.
The base language and library definitions have the same behavior.
The introduction of new modes and routines does not hav an impact
on the compiler and language standards.

When sufficient information is known about a mode
it may be processed entirely at compile time.

The existing mode behavior facility allows specification of
the representation in terms of the mode operator facilities,
which have a fixed representation on the underlying machine.
The expected SPECL facility will allow representation on
the underlying machine to be specified.

E3.

Each program component
will be defined
in the base language,
-in a library, or
in the program.

There will be no default declarations.

.a T
.b

There are no mentions of default declarations in the language.

The explaination following the requirement makes reference
to the translator's default convention.

The compiler accepts declarations which control specific
aspects of how the compilation environment is to be used.
This does not seem to be the notion expressed in the requirment.

<this is not clear -- what does the requirment mean?>

E4.

The user will be able,
within the source language,
to extend existing operators
to new data types.

.a T
.b

The user is able, within the source language, to extend existing operators to new modes..

This is facilitated by the fact that the mode of any object can be detemined using the built-in function MD, and the fact that any routine can be redefined to provide for additional argument modes.

<might mention an operator for doing this redefinition>

E5.

Type definitions
in the source language
will permit definitions of both
- the class of data objects comprising the type and
the set of operations applicable to that class.

A defined type
will not automatically inherit
the operations of the data
with which it is represented.

.a E
.b

The existing mode behavior facility provides for the association of user-defined operations with extended modes.
These operations include
generation, selection, conversion, assignment, printing, and dimensions.

The basic work has been done which shows that this facility may be used to associate any appropriate operations with with the mode.

The behavior of the facility can be defined so that only the desired operatons of the representation will be inherited.

E6.

The data objects
comprising a defined type
will be definable
- by enumeration of their literal names,
as cartesian products of existing types
(i.e., as array and record classes),
by descriminated union
(i.e., as the union of disjoint types), and
as the power set of an enumeration type.

These definitions
will be processed entirely at compile time.

.a E
.b
An enumeration mode operator may be defined using a routine which accepts a list of literal names and produces an appropriate mode with the extended mode facility.

The language has a corresponding set of definitional mechanisms for modes. In conjunction with pointers, they provide the mechanisms necessary to define recursive data structures, and efficient sparce data structures.

<need to find out what a discriminated union is intended to mean,
does it correspond to the genereic modes which means
that once an alternative is chosen for an
instance it cannot be changed, or
does it mean a true union in the sence that
the mode can changed and obsevered (discriminated)>

E7.

Type definitions
by free union
(i.e., union of non-disjoint types) and
subsetting
are not desired.

.a T
.b
Mode definitions by free union and subsetting are not
in the language.

If desired, for some reason, they could be achieved by extension.

<need more details of free union and subsetting>

E6.

When defining a type,
the user will be able to specify the
initialization and
finalization
procedures for the type and
the actions to be taken at the time of
allocation and
deallocation
of variables of that type.

.a E (finialization and deallocation need to be added)
.b
The extended mode facility provides for the specification of initialization using a mode behavior function for generation.

The existing system does not provide for finalization using a mode behavior function for degeneration, this facility could be added if required.

<need some details of Harvard's view of degeneration,
this was disucssed to some extent during last visit>
The mode behavior functions are defined in the encapsulaton housing the rest of the type definition.

<need some details of the requirment's view of encapsulation
does it simply mean the the definitions are contained in the mode, or
does it require that the definitions must be textually contained
in the mode definition
the language provides for textual containment, but
such containment may be as weak as an identifier for
the definition, rather than the definition itself>

F1.

The language will allow
the user to distinguish between
scope of allocation and
-scope of access.

.a E
.b

The language does not have own variables, but this kind of effect can be achieved with the extended mode facility.

<need to get Harvard's view of own variables>

The basic work for controlling access within an allocation has been done.

<need to get more details of what is meant here>

F2.

The ability to limit access to separately defined structures
will be available both
where it is defined and
- where it is used.

It will be possible to associate
new local names with
sepreately defined program components.

.a E
.b
The basic work for access control has been done.

It is possible to associate new local names with
seperately defined program components using a
DECLaration.

<need to get more details of the requirments for access control>

F3.

The scope of identifiers
will be wholly determined at compile time.

.a P/x
.b

The language allows DECLarations to be made anywhere within a block or iteration scope, and multiple use of variable names are allowed in the same scope.

<need to find out more about scope ideas in the requirement>

The scope of identifiers may not necessarily be wholly determined at compile time in the sence that every binding in known. This is the result of the dynamic identification rule for free variables. To the extent that the scope is determineable when a routine is compiled, it is exploited.

The languages handling of free variables provides significant capabilities that a static rule does not provide. The language's scope rule also avoids the problems associated with defining procedures in the scope of other procedures to achieve the correct scope.

<possibly mention the top level envirnment and the fact that this scope rule is natural for an interactive system, and that the apparent overhead can be compiled out when required, except when the additional facilities are needed>

F4.

A variety of
application-oriented data and
operations
will be
available in libraries and
easily accessible in the language.

.a T

.b
The system supports this notion.

<what more can be said>

F5.

Program components
    not defined within the current program and
    not in the base language
will be maintained in libraries
    accessible at compile time.

The libraries
    will be capable of
        holding anything definable in the language and
    will not exclude
        routines whose bodies
            are written in other source languages.

.a E
.b

Program components defined within the current program and not in the base language may be maintained in a library, and accessible whenever required.

The libraries would make use of the general file handling facilities in the language and could hold any object definable in the language.
The existing system does not provide for other source languages, but some capability of this kind could be provided.

Fo.

Libraries and Compools
will be indistinguisable.

They will be
capable of holding anything definable in the language, and
it will be
possible to associate them
with any level of programming activity
from systems,
through projects,
to individual programs.

There will be
many specialized
compools or
libraries,
any user-specified subset of which is
immediately accessible from a given program.

.a E
.b
<what are compools --
a term used in JOVIAL for communication pool
* serves as a central source of data description,
communicating changes in data design by applying
the compiler with the current data description,
parameters, thus allowing alutmatic modification of
references to chaned data in manine language programs.
*/ virtual necessity for large scale systems, where
problems of data design coordination between programers
is to be done>

The requirement seems to simply refer to a facility
for organizing and controlling shared objects.

The existing system provides a number of basic facilities
to provide the required capabilities.

<should say more
simplist -- symbolic files to be LOADed or
DUMPBed files to be LOADBed
would be a way to have a particular environment.
once an envirnment was built up it could also be
SAVEd for later RESTOREs.

the structuring capability could be based on a model of the
way in which the project needs to make use of the contents

subset capability is a mater of allowing selected parts to
be accessed

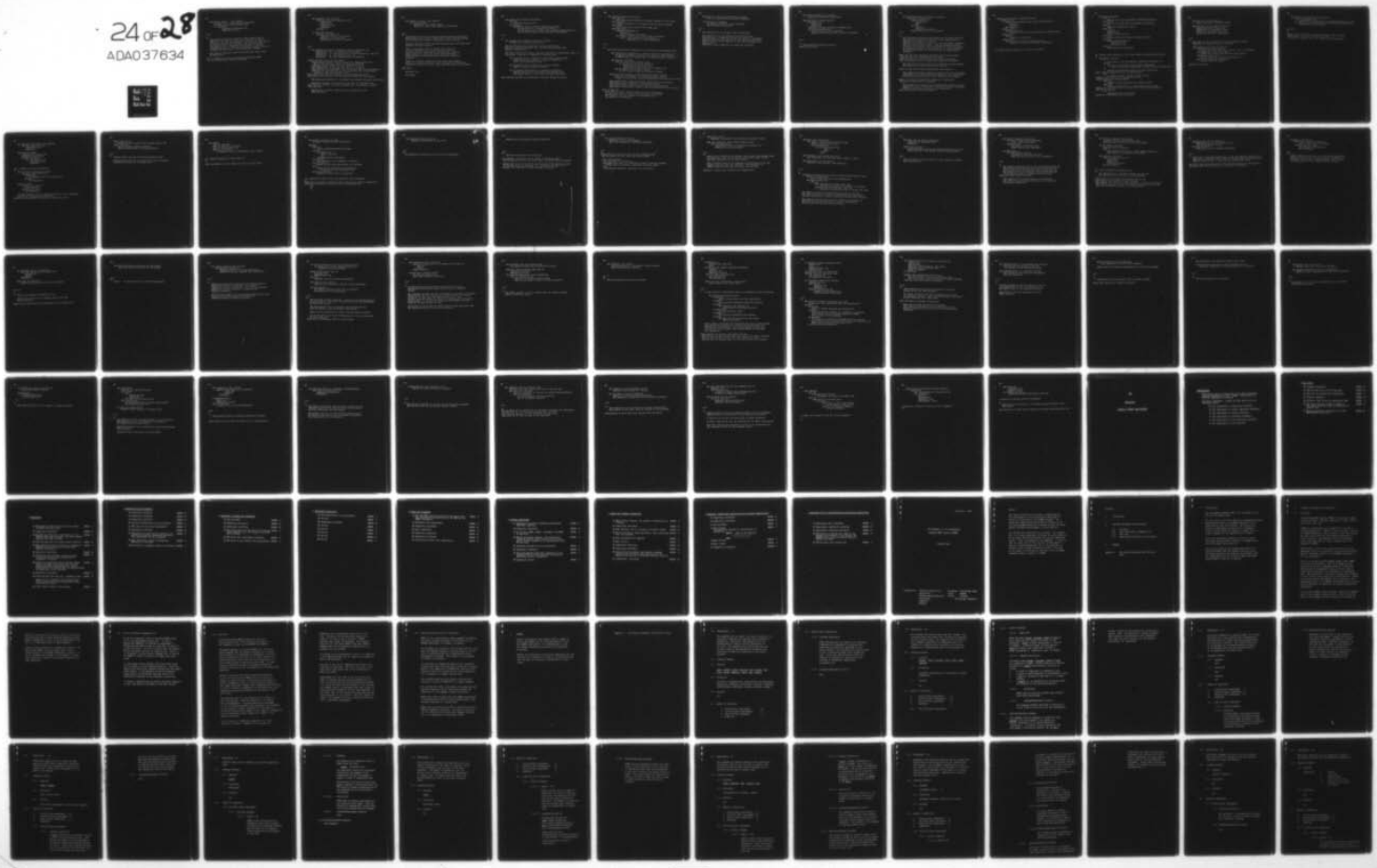
The generallity of the modes of the language and the
support facilities allow these requirments to be satisfied
in a way relevant to the proolem domain.

Solutions satisfying specific requirements could be constructed>

AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

24 OF 28
ADA037634



24 OF 28
ADA037634

F7.

The source language will contain
standard machine-independent interfaces
to machine-dependent capabilities,
- including
peripheral equipment and
special hardware.

.a E

.b

The built-in mode PORT defines a data object used to associate the user's program with a source of input data or a sink for output data. The source or sink may be a terminal device (for instnace, the user's terminal, the printer, or a connection established to another computer system), a file on an external storage device, or a temporary file held in the user's main storage.

This kind of facility could be provided for other kinds of connections.

<is it enough to be able to provide the extended mode, or is a definitonal facility required>

G1.

The language will provided
structured control mechanisms for
sequential,
- conditional,
iterative, and
recursive
control.

It will also provide
control structures for
(pseudo) parallel processing,
exception handling, and
asyncronous interrupt handling.

.a E
.b

The language provides structured control mechanisms for
sequential by a sequence of statements,
conditional by the conditional, exit-conditional, and CASE
interative by the REPEAT block, and
recursive by the recursive use of routines.

<could say more here for each topic
statements may appear in BEGIN blocks or REPEAT blocks and
may be DECLarations, forms, of exit-conditionals
condional a form using the infix operator -> or +>
exit-conditional is the use of => or #> in a block,
CASE provides for explicating the domain and range of conditionals,
iterative is the iterative of the statements in a REPEAT block
possibly with some explicit index
much more can be said to emphasize the exit-conditional and
unification that results, and the notion of mark (<<) and return>

The multi-path facility is the basis for (pseudo) parallel processing.

Exception handling is provided to the user in a uniform way.
<see the TRAP facility write up, and note how its behavior is under
user control>

Asyncronous interrupt handling is an extension of the
TRAP facility.

G2.

The source language will provide
a goto operation
applicable to program labels
- within its most local scope of definition.

.a X
.b

The existing form of the source language does not provide a goto operation to program labels even within the restriction of its most local scope of definition. But, read on ...

An early version of the language provided goto's but they were removed with good reason.

Most of the facilities for conditional execution -- the conditional, iteration, and CASE forms -- serve to provide the power and flexibility generally associated with labels and gotos while retaining clean semantics and encouraging structured, readable programs.

There is a further facility of this sort which allows the user to mark an arbitrary form and then to return from within its evaluation with a user supplied value as its result.

<say more>

The mark (<<)

The RETURN

G3.

The conditional control structures
will be
fully partitioned and
will permit
selection among alternate computations based
on the value of a boolean expression,
on the subtype of a value from a discriminated union, or
on a computed choice among labeled alternatives.

.a X

.b

The conditional statements provide a concise
IF-THEN and IF-ELSE construction;

The exit-conditional statements within a BEGIN-block
provide a unified mechanism for more general constructions
such as IF-THEN-ELSE.

The CASE statement provides a uniform facility to conveniently deal with
situations which occur frequently in programming.

(1) To chose one of a number of conditional alternatives
according to the values of a set of predicates
(i.e., in a decision table).

(2) To choose the Itn action in a set of actions
where I is some integer variable.

(3) To compute the result of a function of several
generic variables in different ways depending on
the actual types of the values of those variables.

<the relation between the requirement and this should be clear>

G4.

The iterative control structure
will permit
the termination condition to appear anywhere in the loop,
will require
control variables to be local to the iterative control,
will allow
entry only at the head of the loop, and
will not impose
excessive overhead in
clarity or
run time execution costs
for common special case termination conditions
(e.g., fixed number of iterations or
elements of an array exhausted).

.a T

.b

The interative control structure is in the form of the REPEAT block.

The termination condition may appear anywhere a loop because:
an exit-conditional statement may appear as any statement in
a REPEAT block (a REPEAT block is the only form of loop).

The control variables
will be local to the REPEAT block
if an explicit index is mentioned
using the FOR modifier and
an initial value which is not shared, and
may be global to the REPEAT block
if the initial value is explicitly SHARED.

Entry is only possible at the head of the loop, because
there is no mechanism for labeling a point within
a loop to which control may be transfered from without the loop.

The common special case termination conditions are explicitly
specifiable using modifiers to the REPEAT block.
This explicitness makes the situation clear and provides
sufficient information so that it may be exploited by the compiler.

<the TO modifier
when used by itself means a fixed number of iterations,
if the array length is known when the loop is compiled,
the compiler could be directed to recognize that
the LENGTH is a constant>.

G5.

Recursive as well as nonrecursive routines
will be available in the source language.

It will not be possible
to define procedures within the body
of a recursive procedure.

.a T
.b

All routines in the language may be recursive.

The scope rule of the language are such that the
declaration of a procedure within the body of a procedure
only provides a local binding for the declared procedure,
with no effect on the binding of any free variables that
may be in it.
Therefore, this requirment is vacuously satisfied.

G6.

The source language will provide
a parallel processing capability.

This capability should include
the ability to
create and
terminate
(possibly pseudo) parrallel processes and
for these processes to
gain exclusive use of resurces
during specified portions of their execution.

.a T

.b
The multi-path control facility.
<more can be said>

G7.

The exception-handling control structure
will permit the user to
cause transfer of
control and
data
for any
error or
exception situation
which might occur in a program.

.a T

.b

All exception conditions are handled by the TRAP facility.
Any trap that the system detects in some scope may be
handled by the user if desired.
The user may associate a form with an identifier associated
with the trap to be handled. When the situation occurs
the form is evaluated and the environment may be examined
to determine details of the situation. If the situation
is correctable then the correction may be computed and
returned.

<more can be said, including the use of the BREAK and CONT
facility with the explaination capability,
also the fact that the facility is written in ECL and
may be extended, give the exceptions currently provided>.

The user is able to get out of an arbitrary control context
and intercept at any embeding level desire.

<this is done using marked expressions and returns>

The exception-handling mechanism permits the user to specify
the action to be taken upon the occurrence of a designated
exception within any given access scope of the program.

<this is a result of having an appropriate identifier
declared in the scope of interest>

The transfers of control may be effectively forward in the
program or out of a procedure through its calling structure.
(there is no lexical structure for procedures)
<this is the use of mark and return>

G8.

There will be source language features
which permit
delay on any control path until some specified
time or
situation
has occured,
which permit
specification of
relative priorities among parallel control paths,
which give
access to
real time clocks, and
which permit
asynchronous hardware interrupts to be
treated as any other exception condition.

.a T (multi-path, extension of interrupt handler)

H1.

The source language
    will be
        free format with an explicit statement delimiter,
    will allow
        the use of memonically significant identifiers,
    will be
        based on conventional forms,
    will have
        a simple,
        uniform, and
        easily parsed grammar,
    will not
        privide unique notations for special cases,
    will not
        permit abbreviation of
            identifiers or
            key words, and
    will be
        syntactically unambiguous.

.a T (with minor exception of the BEGIN block abreviated delimiters)
.b
The source language
    is
        free format with the explicit statement delimiter ";",
    allows
        the use of memonically significant identifiers
            (the two kinds of identifiers without length restriction),
    is
        based on conventional forms with some extensions,
<the classical appearance with generalization>
    has
        a simple, uniform, easily parsed grammar,
<simple and uniform by the notion of a form,
 easily parsed by LALR(1) parser>
    does not
        provide unique notations for special cases,
    does not
        permit abbreviations of identifiers or key words,
<except for the BEGIN ("[)") and END ("(]") which are convienant
 but not necessary>
    is
        syntacitically unambiguous.
<based on LALR parser and the window>

H2.

The user will not be able to
modify the source language syntax.

Specifically, he will not be able to
modify operator hierarchies,
introduce new precedence rules,
define new key word forms, or
define new operator precedences.

.a X
.b
The user is able to modify the source language syntax,
although this capability may be limited.

Specifically, the user is able to
modify operator hierarchies
using the fixity operators PREFIX, INFIX, and SYNFIX,
introduce new precedence rules
using INFIX which also controls associtivity,
define new key words
using SYNFIX or redefine the parser, or
define new operator precedences
using INFIX and SYNFIX.

<more can be said>

H3.

The syntax of source-language programs
will be composed from
a character set suitable for publication purposes, but
no feature of the language will be inaccessible using the
64-character ASCII subset.

.a T (?)
.b

<need to know what the 64-character ASCII subset contains
worry about "\" and the "{","}", "<",">",
if this is a problem then just change some defintions?>

H4.

The language definition will provide
the formation rules for
identifiers and
- literals.

These will include
literals for numbers and
character strings and
a break character
for use internal to
identifiers and
literals.

.a T (except for break characters)
.b
The language definition provides
the formulation rules for
identifiers
as <describe the two flavors> and
literals
as <describe>.

These include
literals for numbers
as <describe>
character strings
as <describe>.

A break character is not included, but the "\" is available
for use in alphanumeric identifiers.
<could have some other definition by redefining LEX>.

H5.

There will be no
continuation of lexical units across lines, but
there will be a
way to include object characters
such as end-of-line in literal strings.

.a T
.b

Lexical units may not be continued across lines.

Object characters such as end-of-line may be included
in literal strings by including them.

H6.

Key words
will be reserved,
will be very few in number,
will be informative, and
will not be usable
in contexts where an identifier can be used.

.a T (their definition of key words...)
.b

<see the appendix in the manual for list of words used>

H7.

The source language will have
a single, uniform comment convention.

Comments
will be
easily distinguishable from code,
will be
introduced by
one or possibly
two
language-defined character,
will permit
any combination of characters to appear,
will be able to
appear at any reasonable point in a program,
will
automatically terminate at end-of-line
if not otherwise terminated, and
will not prohibit
automatic reformatting of programs.

.a T (exception taken to the line oriented, card oriented)
.b
<describe the comment convention that exists in the form or operators>
<some other convention could be used by redefining LEX>
IE
/*
*/

H8.

The language will not permit
unmatched parentheses of any kind.

.a T
.b
No unmatched parenthesis of any kind are permitted.

H9.

There will be a uniform referent notation.

.a E
.b
This may be achieved by extension.

<a procedure application of an object of arbitrary mode
may be defined so that selection looks like procedure application

before this is done, however, some consideration should be given
to the ECL notion of selection as a function, in fact it is
defined this way even for user defined selection >

H10.

No language-defined symbols
appearing in the same context
will have essentially different meanings.

.a T
.b
<consideration should be given to the advantages of
the ECL notion of pure value and proper object

also, in particular
= is not used for assignment,
parameters are forms and unnecessary parens have not meaning,
<infact they are removed, including the outer ones>
but,
polymorphic operators (generic) may be defined.

11.

There will be no
defaults in programs which affect the program logic.

That is, decisions which affect program logic
will be made either
irrevocably when the language is defined, or
explicitly in each program.

.a T
.b

There are no defaults in programs which affect the program logic.
All decisions which affect program logic are either in the
language definition or in each program.

What a program does in the language is determinable from the
program and the model for language. Each decision is
either global to some scope or local to each use.

<nothing is known that violates this requirement>

Defaults will be provided
for special capabilities affecting only
object representation and
other properties
the programmer does not
know or
care
about.

Such defaults will always mean that
the programmer does not care which choice is made.

The programmer will be able to
override these defaults when necessary.

.a T

.b
Defaults are provided for special capabilities affecting only
object representation
by the mode constants and constructors,
and other properties
such as
open and closed routines which are
semantically equivalent (default is closed),
reentrant and nonreentrant which are
subsumed by reentrant which is always the case.

The SPECL facility will privide the ability to override
the default choices of object representation and management
and code generation to exploit specific architectural features.

The existing ECL compiler facility includes the ability to
override the default choices or variable-invariencies
and rewriting of specific routine usages.

13.

The user will be able to associate compile-time variables with programs.

These will include variables which specify the object computer model and other aspects of the object machine configuration.

.a E
.b

With the SPECL facility specific object machine features may be exploited.

14.

The source language will permit
the use of conditional statements
(e.g., case statements)
dependent on
the object environment and
other compile-time variables.

In such cases,
the conditional will be
evaluated at compile time and
only the selected path will be compiled.

.a T

.b
The language provides the use of conditional statements such as the CASE statement which may be dependent on the environment of compilation. The general facility for compiling only the selected path is provided for in the existing compile through computed macros.
<say more about computed macros>

This facility is a special case of the facility for identifying constant expressions and evaluating them during compilation.

The source language will contain
a simple, clearly identifiable base
which houses all the power of the language.

To the extent possible,
the base will be minimal,
with
each feature providing a single unique capability
not otherwise duplicated in the base.

The choice of the base
will not detract from the
efficiency,
safety, or
understandability
of the language.

.a T (with pragmatic considerations)
.b
The language EL1 is the base language of ECL and
as such houses all the power of the language.

<more could be said about the choices made in EL1
particularly the pragmatic considerations
for example, one does not define iteration in terms of recursion
also, SPECL will provide for the explicit factoring of the EL1
into the components of which it is constructed>

16.

Language restrictions which are
dependent only on the translator and
not on the object machine
will be specified
explicitly in the language definition.

.a T
.b
There are no language restrictions which are dependent only on the translator except for those which may be dependent on the underlying machine features such as word size or address space size.

<at least none are documented in the reference manual for the compiler and none have been encountered in the use of the system>

17.

Language restrictions
which are inherently dependent
only on the object environment
will not be built into
the language definition or
any translator.

.a T
.b

These restrictions are not in the existing facilities, and the SPECL facility will provide for making the required reports when a program exceeds the resources or capabilities of the intended object machine.

J1.

The language and its translators
will not impose run time costs for
- unneeded or
- unused
generality.

They will be capable of
producing efficient code for all programs.

.a P, S

.b

This is the general philosophy of ELI.

It will be carried out in greater detail with the
SPECL facility.

<more can be said about the elaboration of the requirement>

J2.

Any optimizations performed by the translator will not change the effect of the program.

.a T
.b

This is the basic notion of a valid optimization.

J3.

The source language will provide
encapsulated access to
machine-dependent hardware facilities,
including machine language code insertions.

.a T

.b

Machine language code insertions are encapsulated as hand coded explicit procedure. The external appearance of such a procedure, in terms of parameter passing and result returning, is that of the explicit procedure.

This facility along with the extended mode facility may be used to provided encapsulated access to machine-dependent hardware facilities.

J4.

It will be possible within the source language
to specify the object representation of
composite data structures.

These descriptions will be
optional and
encapsulated and
will be
distinct from the logical description.

The user will be able to
specify the time/space tradeoff to the translator.

If not specified,
the object representation will be optimal,
as detemined by the translator.

.a S
.b

The existing facility provides a choice for the representation of all objects. With SPECL the user will have the opportunity to make this choice.

The descriptions will be optional and encapsulated and will be distinct from the logical description.

There will be provision to specify the time\space tradeoff.

If not specified the object representation will be determined by the translator.

<more SPECL information would be useful here>

J5.

The programmer will be able to
specify whether calls on a routine are to have an
- open or
- closed
implementation.

An open and a closed routine
of the same description
will have identical semantics.

.a E

.b
A routine mayoe sepecified to the compiler as being a substitution macro which means that it will be compiled inline.

The language provides for the equivalence of the open and closed forms except for the possibility of side effects if a formal parameter is used more than once in the body. This problem could be avoided in general by additional intermediate declarations, or in specific cases if more analysis of the usage of an open routine is done.

The choice to being open or closed could be made automatic when the analysis tools of the PMS are available.

K1.

The language will not require that
the object machine have an operating system.

When the object machine does have an
operating system or
executive program,
the hardware/operating sysem combination
will be interpreted as
defining an abstract machine which
acts as the object machine for the translator.

.a S

.b
The SPECL facility will not require that the object machine
have an operating system.

K2.

The language will support
the integreation of seperately written modules
into an operational program.

.a T
.b

This is provided by the link facility.

K3.

A family of
programming tools and
aids
in the form of support packages including
linkers,
loaders, and
debugging systems
will be made available with
the language and
its translators.

There will be a consistent, easily used
user interface for these tools.

.a T
.b
A rich family of programming tools is available in the ECL system.

The support packages include
a compiler
which is compatible with the interpreter,
a linker
for combining seperately compiled packages,
a prober
for observing the behavior of
interpreted or compiled routines,
a backtracer
for observing the stack,
a breaker
for setting breakpoints and steping,
a traper
for defining the behavior of the system
when traps occur.

All of these facilities are consistent and easily used through
the interactive environment provided by the top level and
break level environments. These environments differ only
in that the stack defined when a break occurs is available
for inspection.

<more details to include more facilities and
the notion that system facilities are unified with user facilites
such as LEX, PARSE, COMPILE, COMPILE, CONSTRUCT, etc...
and the lack of special times ... see some of the early papers>

K4.

A variety of useful options to aid
generation,
test,
documentation, and
modification
of programs will be provided as
support software available with
the language or
as translator options.

As a minimum, these will include
program editing,
post-mortem
analysis and
diagnosis,
program reformating for
standard indentations and
cross-reference generation.

.a T
.b
The system's support facilities which aid
generation, test, documentation, and modification
includes
an editor
which is syntax directed and interactive,
a prober
for interactive analysis and diagnosis of problems
which may be used to produce execution traces
at the level of the language,
an unparser
for mapping the abstract representation of programs
to a parsable and formated (pretty-printed) concrete form
which has a standard indentation rule,
a cross-reference generator.

K5.

The source language will permit inclusion of
assertions,
assumptions,
axiomatic definitions of data types,
debugging specifications, and
units of measure
in programs.

Because many assertional methods are
not yet powerful enough for practical use,
nor sufficiently well developed for standardization,
they will have the status of comments.

.a E
.b

These facilities may easily be included using the facilities of the base language.

An ASSERT routine exists and the others may be done in a similar fashion. Their interpretations will depend on the evaluator that being used.

<see the paper of multiple evaluators>

They may be given the status of comments.
This kind of information will be exploited by the Program Manipulation System which is currently being developed.

L1.

No implementation of the language will contain source-language features which are not defined in the language standard.

Any interpretation of a language feature not explicitly permitted by the language will be forbidden.

.a T

.b

A basic concept of the ECL system is that the all implementations be a refinement of the ECL model which is the language standard.

This concept will be realizable with the SPECL facility.

L2.

Every translator for the language
will implement the entire base language.

There will be no subset implementation of the base language.

.a T

.b

Translators for the language will be based on SPECL.

<some more details are required on SPECL>

L3.

The translator will minimize compile time costs.

A goal of any translator for the language will be low-cost translation, (when optimizaion is disabled).

.a T

.b

Use of optimization is optional.

L4.

Translators will be able to
produce code for a variety of machines.

The machine-independent parts of translators
might be built independently of the code generators.

.a S

.b
Translators for a variety of machines will be provided
by the SPECL facility.

L5.

The translator need not be able to
run on the object machine.

Self-hosting
is not required, but
is often desirable.

.S

.b

The SPECL facility will be capable of cross compiling.

L6.

The translator
will do full syntax checking,
will check
- all
operations and
parameters
for type compatibility, and
will verify that
all language-imposed semantic restrictions
on the machine are met.

It will not automatically
correct errors detected at compile time.

.a T

.b

The compiler for the language operates on the abstract representation which is produced by the parser for syntactically correct forms.

Routine parameters may be checked for type compatibility when compiled.

Automatic error correction is not provided.

L7.

The translator will produce
compile time explanatory diagnostic
error and
warning
messages.

A suggested set of
error and
warning situations
will be provided
as part of the language definition.

.a T

.b
The existing compiler produces diagnostic messages.

<more should be said about the specifics in explainations>

L8.

The characteristics of translator implementations will not be dictated by the language definition or standards.

.a T

.b

The compiler and other facilities are written in the language itself and as such are more able to evolve with new technology and techniques.

One example of this is the advanced analysis tools of the PMS which may be used to collect information for exploitation by the compiler.

L9.

Translators for the language will be written in their own source language.

.a T

.b

The existing compiler is written in its own source language and has been used to compile and improve itself.

M1.

The language will be composed from
features which are within the state of the art and
any design or redesign
- which is necessary to achieve the needed characteristics
will be conducted
as an engineering design effort and
not as a research project.

.a T

.b
<the language has the potential for absorbing aparently new technology
in fact some of the features have been used to model
new features apearing in the literature in such a way
that they may be used as part of the language>

M2.

The semantics of the language will be
defined unambiguously and clearly.

To the extent a formal definition
assists in attaining these objectives,
the language's semantics will be specified formally.

.a T

.b
The semantics of the language are defined unambiguously
and clearly by the model which is written in the language.

<more should be said about this approach and the model>

M3.

The user documentation of the language will be
complete and
will include both
- a tutorial introductory description and
a formal in-depth description.

The language will be defined
as if it were the
machine-level language of an
abstract digital computer.

.a P

.b

There currently exists a reference manual which is reasonably complete and in the process of being refined and updated.

A tutorial is in form of class notes is being developed.

A formal description will be contained in the model descriptions.

The model defines the language in terms of an interpreter for the language written in the language itself.

M4.

The language
will be
configuration-managed
throughout its total life cycle and
will be
controlled at the DoD level
to ensure
that there is only one version
of the source language and
that all translators conform
to that standard.

.a <this is a policy in the use of the language>

.b

M5.

There will be identified support agent(s)
responsible
for maintaining the translators and
- for associated
design,
development,
debugging, and
maintenance aids.

.a <this is a policy in the use of the language>
.b

M6.

There will be
standards and
support agents
for common libraries,
including application-oriented libraries

.a <this is a policy using the language>

.b
This kind of requirement is certainly easily achievable with
the system.

<see some of the past Harvard papers on package specialization, etc>

# LTR

# EVALUATION

# TOWARDS "TINMAN" REQUIREMENTS

## INTRODUCTION

This report gives an evaluation of the LTR programming language compared with the " TINMAN" requirements of the U.S. Department of Defense.

For each requirement, a grade has been given with the following scale:

10: The requirement is completely satisfied.

8: The requirement is almost completely satisfied.

6: The requirement is mostly satisfied.

4: The requirement is partially satisfied.

2: The requirement is very partially satisfied.

0: The requirement is not satisfied.

## A DATA TYPES

| | | |
|---|---|---|
| A1 | Complete agreement | GRADE: 10 |
| A2 | LTR provides also a bit string type | GRADE: 10 |
| A3 | LTR does not provide that possibility | GRADE: 0 |
| A4 | Complete agreement | GRADE: 10 |
| A5 | Character type is not an enumeration type | GRADE: 2 |
| A6 | The lower subscript bound is always 1 the upper subscript bound is determined at compile time | GRADE: 6 |
| A7 | This is completely satisfied by the LTR EQUIVALENCE declaration | GRADE: 10 |

B OPERATIONS

B1 Satisfied but LTR does not allow to define to define new data types — GRADE: 8

B2 Completely satisfied — GRADE: 10

B3 The LTR type defined by enumeration is the QUALITY type. For that type, the only allowed operators are EQL and NEQ. — GRADE: 8

B4 LTR provides also an integer modulo operator. — GRADE: 10

B5 LTR truncation rules are correct. Rounding is implicit for floating point-fixed point numbers conversions. — GRADE: 8

B6 Completely satisfied. — GRADE: 10

B7 LTR allows data transfer between records. It does not allow scalar operations or assignment on arrays. — GRADE: 4

B8 There is no implicit type conversion except between integer, fixed point and floating point data.Conversion between the object representation of numbers and the representation as characters is not provided. — GRADE: 6

B9 Completely satisfied. — GRADE: 10

B10 LTR provides real-time I/O . Dynamical assignment is not provided. The I/O operations are partially installation-dependant. There is no file facility. — GRADE: 8

B11 That feature seems to be missing. — GRADE: 0

## C EXPRESSIONS AND PARAMETERS

| | | |
|---|---|---|
| C1 | Completely satisfied | GRADE: 10 |
| C2 | Completely satisfied | GRADE: 10 |
| C3 | Completely satisfied | GRADE: 10 |
| C4 | Constant expressions are not provided. | GRADE: 0 |
| C5 | LTR seems to meet that requirement. | GRADE: 8 |
| C6 | Completely satisfied | GRADE: 10 |
| C7 | Parameters by value and by reference are provided. Exception parameters and procedure parameters are not available. | GRADE: 8 |
| C8 | Type, precision, scale are mandatory. Dimension is optional. | GRADE: 4 |
| C9 | The number of arguments cannot be variable. | GRADE: 0 |

D VARIABLES, LITERALS AND CONSTANTS

| | | |
|---|---|---|
| D1 | Not provided. | GRADE: 0 |
| D2 | Completely satisfied. | GRADE: 10 |
| D3 | Completely satisfied. | GRADE: 10 |
| D4 | The range cannot be specified for floating point numbers, only for integer and fixed point numbers. | GRADE: 8 |
| D5 | LTR meets that requirement partially. | GRADE: 4 |
| D6 | LTR seems to meet mostly that requirement. | GRADE: 6 |

E DEFINITION FACILITIES

| | |
|---|---|
| E1 That possibility is not provided. | GRADE: 0 |
| E2 (Cf E1) | GRADE: 0 |
| E3 Completely satisfied. | GRADE: 10 |
| E4 (Cf E1) | GRADE: 0 |
| E5 (Cf E1) | GRADE: 0 |
| E6 (Cf E1) | GRADE: 0 |
| E7 (Cf E1) | GRADE: 10 |
| E8 (Cf E1) | GRADE: 0 |

F SCOPE AND LIBRARIES

| | | |
|---|---|---|
| F1 | The LTR SETS provide controlled allocation data. The corresponding primitives are the NEW and the ERASE statements. | GRADE: 8 |
| F2 | LTR meets that requirement. | GRADE: 8 |
| F3 | Completely satisfied. | GRADE: 10 |
| F4 | Well fullfilled. | GRADE: 8 |
| F5 | Completely satisfied. | GRADE: 10 |
| F6 | Completely satisfied. | GRADE: 10 |
| F7 | LTR fulfills mostly that requirement. | GRADE: 6 |

G CONTROL STRUCTURES

| | | |
|---|---|---|
| G1 | Completely satisfied including asynchronous interrupt handling | GRADE: 10 |
| G2 | Completely satisfied. | GRADE: 10 |
| G3 | In an "IF...THEN...ELSE" statement, the ELSE is optional. | GRADE: 8 |
| G4 | Mostly satisfied. However, the termination condition cannot appear anywhere in the loop and a control variable is not necessarily local to the loop. | GRADE: 8 |
| G5 | Recursive procedures are not provided. | GRADE: 0 |
| G6 | Completely satisfied. | GRADE: 10 |
| G7 | That requirement seems well satisfied by the standard LTR interrupt handling (provided the operating system is adequate). | GRADE: 8 |
| G8 | Completely fitted. | GRADE: 10 |

H SYNTAX AND COMMENT CONVENTIONS.

| | | |
|---|---|---|
| H1 | Well fitted. However, the length of identifiers is limited. | GRADE: 8 |
| H2 | Completely satisfied. | GRADE: 10 |
| H3 | LTR character set is a subset of 64 ASCII subset | GRADE: 10 |
| H4 | Mostly satisfied. Long identifiers (with underline) are not allowed. | GRADE: 6 |
| H5 | Card boundaries are ignored. | GRADE: 0 |
| H6 | Well fitted. | GRADE: 8 |
| H7 | Completely satisfied. | GRADE: 10 |
| H8 | Completely satisfied. | GRADE: 10 |
| H9 | Array referent notation and function referent notation are identical. Data selection by the mean of pointers has a different referent notation. | GRADE: 6 |
| H10 | Completely satisfied. | GRADE: 10 |

## I DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

I1 Completely satisfied. GRADE: 10

I2 Completely satisfied. GRADE: 10

I3 Not provided. GRADE: 0

I4 Well fitted. GRADE: 8
A conditional sequence of instructions is defined by:

```
$BEGIN   name of the sequence
.....................
.....................
$END
```

I5 Well fitted GRADE: 8

I6 Well fitted GRADE: 8

I7 Completely satisfied. GRADE: 10

## J EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

| | | |
|---|---|---|
| J1 | Requirement well satisfied. | GRADE: 8 |
| J2 | Requirement completely satisfied. | GRADE: 10 |
| J3 | Requirement completly satisfied. | GRADE: 10 |
| J4 | LTR allows to specify the order of the fields and the width of the fields. The LOGICAL type allows to specify multiple fields in a word. | GRADE: 8 |
| J5 | LTR provides only closed call. | GRADE: 0 |

Reference: S656

# An Assessment of the Programming Language MORAL against TINMAN

October 1976

Prepared by: Software Sciences Ltd
Abbey House
282-292 Farnborough Road
Farnborough
Hampshire
England

Telephone: Farnborough 44321
Telex: 858228
Cables: Software
Farnborough Hampshire

## Preface

This report presents an assessment of MORAL against the requirements of the United States Department of Defence, as described in the TINMAN document. The report was commissioned from Software Sciences Ltd by the United Kingdom Ministry of Defence through the Royal Signals and Radar Establishment at Great Malvern.

The contents are essentially in two parts. The largest part is a formal assessment following the lines presented in the paper "Evaluation Methods for the HOL Comparison Effort". This is contained in the Appendix. It was felt that the sheer size of this section is something of a barrier to understanding and that it was important to include a more informal discussion as well. The body of the report is therefore an overview of how the MORAL language stands in respect to TINMAN.

Contents

1. Introduction

2. Informal Assessment and Discussion

2.1 Attitudes
2.2 Level and General Language Style
2.3 Real Time
2.4 Other Particular Points of Difference

3. Summary

Appendix: The Formal Assessment and Evaluation Chart

## 1. Introduction

The programming language MORAL is a development which has had two primary aims.

The first was to provide a high level programming medium geared specifically to the MASCOT approach to the production of real-time systems. The second was to provide a high degree of portability of programs within the UK through the adoption of CORAL 66 as a target language for the implementation of MORAL.

Within the constraints imposed by these two goals an attempt has been made to produce a language which actively supports good structure.

The initial design of the language was carried out between November 1975 and January 1976 and a first prototype translator was completed in September 1976 for experimental use at the Royal Signals and Radar Establishment at Malvern in England.

## 2. Informal Assessment and Discussion

### 2.1 Attitudes

In assessing MORAL against TINMAN we have had to adopt certain attitudes. The most important of these concerns our view of real time.

TINMAN is conventional in its approach to time and to task synchronisation. MORAL, on the other hand, reflects the MASCOT approach to the structure of real time systems. We take the attitude that this approach represents a valid solution to the problem of real time structure and in some ways offers a better approach than that assumed by TINMAN.

Whatever one's view on the relative merits of the two approaches they are so different that detailed discussion against individual aspects of the TINMAN requirement is not very meaningful.

There is a second area where MORAL departs from TINMAN on a matter of principle. This is in respect of the question of how far a language should go in defining such details of a program's execution as the order of evaluation of operands in an expression or a parameter list. We believe that it can never be good for a program to rely for its correct operation on such details. Indeed we would go further and suggest that it would be a not unreasonable test of a programs correctness that it should survive deliberate variation in the way such matters are implemented.

In the case of MORAL there is another reason for stopping short of such complete specification. Since one of the aims of the language was that translation into CORAL 66

should be a practical and effective road to portability the definition had to avoid imposing constraints which might be impossible to meet in any straight-forward way due to the existing variety of CORAL compilers.

These two areas account for a significant number of the points where MORAL fails to comply with a strict interpretation of the TINMAN requirement. However, we do not recommend any changes to the language to cover these cases as we regard the principles involved as of some importance.

## 2.2 Level and General Language Style

As far as its general style is concerned MORAL seems to fit the requirements rather well. It has an appropriate emphasis on allowing the programmer to develop and use abstractions in a way that is explicit in what he writes. It tries to ensure that the compiler detects the great majority of simple errors. While it is altogether a rather smaller language than is implied by TINMAN its basic structures are modern and fairly clean.

It lacks some of more powerful facilities which would have increased its cost disproportionately. These are compile time facilities, operator definitions and dynamic array bounds. It deliberately avoids over-complete specification of evaluation order and fails to be sufficiently flexible in the treatment of precision.

It offers a comprehensive and rather individual approach to real time which is discussed in the next section.

## 2.3 Real Time

As mentioned above MORAL treats real time in a very particular way. This reflects the origin of the language as a part of the MASCOT development.

The MASCOT approach to the development of a real time system demands that the system be defined as a network whose elements are independently schedulable activities and data areas such that there is no other interraction between different activities than the consequences of their common access to particular data areas. All such access is explicitly shown in a network diagram and provides the focus for all time-sensitive or synchronising behaviour.

Within an activity there are no parallel operations. In fact the facilities of MORAL are at two levels in respect of this structure. Primitives are provided at quite a fundamental level for controlling the access to the common areas while higher level encapsulation facilities are there to allow all dependence on the primitives to be completely protected and hidden from the bodies of the activities.

This approach aims for true modularity of software in real time systems, allowing components to be reusable and interchangeable. A second significant advantage is that the network diagram which represents the structure of the system is largely independent of the number of processors employed and changes to the computer configuration can be implemented without significant change to the software structure.

In the context of TINMAN this approach is a little difficult to assess. TINMAN is written under the

assumption of a conventional structure for real time. The most critical area which needs to be looked at is that of error recovery. TINMAN requires that normal 'ON Statement' facilities be available for this while MASCOT (and hence MORAL) demands that all non sequential possibilities are designed into the network.

In looking at the consequences of this it is important to distinguish between the two types of use which are made of ON Statements.

One type is as a way of coping with an actual error condition of some kind. The other is as part of the application logic where some of the scheduling is driven by events.

Using MORAL the first type of use is the only one which is relevant and the attitude adopted here is that for errors which are only detectable by hardware (i.e. those for which an ON statement is the only way of trapping than in the program) these will either be associated with peripheral activity and handleable in the normal way, or will necessitate shutting down and restarting part of the network. This last capability needs to be designed in at the network level and not left to individual programmers.

## 2.4 Other Particular Points of Difference

MORAL is not a particularly large language and besides the points already discussed it falls short of the TINMAN requirement in a number of respects.

Its treatment of alternative record types does not quite meet the standard required. The deficiency is in the level of security which can be guaranteed without unacceptable overheads. This aspect of the language is likely to be revised.

Its treatment of enumeration types is not as general as would have been liked and offers scope for extension. However, the suggestion in TINMAN that character sets should be able to be treated as any other enumeration set is probably too large a step to take.

The available precision for numeric calculation is limited by the use of CORAL 66 as a target language.

The related facilities of booleans, bit operations and powerset types are not all available although the capability of the language is quite sufficient.

MORAL goes rather further than the TINMAN requirements in providing facilities for restricting access to the internal structure of a record type.

MORAL does not provide any form of controlled access to machine dependent facilities. Nor does it allow the user to control the way particular language features are to be implemented in individual cases.

## 3. Summary

Overall we would say that despite quite a number of points where MORAL fails in varying degrees to meet the TINMAN requirements, the language goes a long way towards satisfying the needs of the Department of Defence.

Whether the differences of principle regarding real-time structure are acceptable is a matter for discussion and this can only be resolved by consideration of wider issues.

Appendix - The Formal Assessment and Evaluation Chart

1.0. Requirement: A1

The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable, and component of composite data structures be explicitly specified in the source programs.

2.0. Language Elements

2.1. Keywords

type, integer, fixed, floating, byte, status, ref, array, group, endgroup, const, case, endcase

2.2. Structures

arraytype, assignabletype, constanttype, declarabletype, directtype, paramtype, pointertype, proctype, rangetype, referencetype, simpletype, typedec, usertype, casedec

2.3. Features

N/A.

3.0. Degree of Compliance

| | | |
|---|---|---|
| 1. | Totally meets requirement | (X) |
| 2. | Partially meets requirement | ( ) |
| 3. | Does not meet requirement | ( ) |
| 4. | Undefined | ( ) |

## 3.1. Totally meets Requirement

### 3.1.1. Language Description

MORAL requires that the types of all variables, parameters and components of composite data structures be explicitly specified in the source program. This includes full specification of the alternative formats that records may take. These type specifications enable the compiler to perform full compile-time type checking of parameters, expressions, assignments etc.

### 3.1.2. Language/Requirement Conflict

None

1.0. Requirement: A2

The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (ie. composite data structures with indexable components of homogeneous type) and records (ie. composite data structures with labelled components of heterogeneous type) as type generators.

2.0. Language Elements

2.1. Keywords.
integer, fixed, floating, byte, array, group, endgroup

2.2. Structures

arraytype, groupdefinition, simpletype, typedec, typedefiner.

2.3. Features

N/A

3.0. Degree of Compliance.

| | | |
|---|---|---|
| 1. | Totally meets requirement | ( ) |
| 2. | Partially meets requirement | (X) |
| 3. | Does not meet requirement | ( ) |
| 4. | Undefined | ( ) |

3.2. Partially meets requirement.

3.2.1. Current Language.

3.2.1.1. Degree:90%.

MORAL provides integer, floating, fixed and byte as basic types and both arrays and groups as type generators. (group in MORAL corresponds to 'record' in TINMAN). In addition MORAL provides ref and status as further type generators. (status in MORAL corresponds to 'enumeration sets' in TINMAN).

3.2.1.2. Language Description.

The Simple types integer, floating, fixed and byte are predefined in MORAL. Further types may be defined by means of a typedec which defines a new type as either:

1. an array of components of homogeneous type,
2. a group of named components of heterogeneous types,
3. a ref to a variable of some type (i.e. a typed pointer)
4. a status (ie. an enumeration of the values taken by variables of the type being defined.)

3.2.1.3. Limitations

MORAL does not provide a Boolean type distinct from other simple types.

3.2.1.4. Language/Requirement Conflict.

The language elements specified in subsection 2 do not conflict with any of the HOL requirements.

3.2.2. Modified/Extended Language.

The language could be extended to include the type boolean; this would appear in the syntax as a further alternative of the syntax production 'Simpletype'. At present, boolean expressions may only appear in restricted contexts (eg. if then);

further, certain boolean operations are defined on numeric data. Any extension to include booleans would require a rationalisation and integration with these and other features, eg. B6 (Boolean operations), B11 (powersets).

1.0. Requirement: A3.

The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

2.0. Language Elements.

2.1. Keywords.

none.

2.2. Structures.

none.

2.3. Features.

N/A.

3.0. Degree of Compliarce.

1. Totally meets requirement ( )
2. Partially meets requirement ( )
3. Does not meet requirement (X)
4. Undefined

3.3. Does not meet requirement.

3.3.1. Current language.

3.3.1.1. Reasoning

Although MORAL allows specifications of the precision and range of fixed point and integer variables, it does not allow specification of the precision of floating point arithmetic or for individual variables.

### 3.3.2. Modified/Extended Language.

Extensions could be made to the language to meet this requirement. Indeed this could be considered virtually independantly of other language features. However no suggestion is made here for the format of this extension as this would require the resolution of several problems: eg. whether the precision refers to stored values or to operations or both; effects of assignment between variables of different precisions, etc.

1.0. Requirement: A4.

Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.

2.0. Language elements.

2.1. Keywords:

fixed, integer.

2.2. Structures:

Size, Fraction bits.

2.3. Features

Scale factor management is done by the compiler.

3.0. Degree of Compliance

| | | |
|---|---|---|
| 1. | Totally meets requirement | (X) |
| 2. | Partially meets requirement | ( ) |
| 3. | Does not meet requirement | ( ) |
| 4. | Undefined | ( ) |

3.1. Totally meets requirement.

3.1.1. Language Description

A fixed variable must be declared to have a certain size and Fraction bits. Size defines the number of bits allocated to storing the number and therefore defines its range. Fractionbits defines how many bits are regarded as the fractional bits, and therefore defines the step size.

Both Size and Fractionbits are integer quantities determined at compile time. Note that Fractionbits may be positive or negative and may have a value such that the binary point falls outside the significant field of the number.

3.1.2. Language/Requirement Conflict.

None

1.0. Requirement A5

Character sets will be treated as any other enumeration type.

2.0. Language elements.

2.1. Keywords

status

2.2. Structures.

Simpletype.

2.3. Features.

N/A

3.0. Degree of Compliance.

3.2. Partially meets requirement.

3.2.1. Existing Language.

3.2.1.1. Degree: 70%

MORAL allows the definition of enumeration sets which includes the definition of program literals and the order of the elements. These properties are unalterable at run-time.

3.2.1.2. Language

The definition of enumeration sets in MORAL takes the form:

status (Identifierlist)

The identifers listed are the program literals for the members of the enumeration set and are assumed to appear in order of increasing value.

Additionally, the built-in simple type byte is defined to take values which are the positive integer interpretations of the character set of the execution environment.

3.2.1.3. Limitations

MORAL does not define input/output of the values of basic types. Thus the effects of transferring a status value to or from a devices are not defined.

3.2.1.4. Language/Requirement Conflict.

None

3.2.2. Modified/Extended Language

None proposed.

1.0. Requirement: A6

The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

2.0. Language Elements.

2.1. Keywords.

array

2.2. Structures.

Arraytype, Range

2.3. Features

N/A

3.0. Degree of Compliance.

1. Totally meets requirement ( )
2. Partially meets requirement (X)
3. Does not meet requirement ( )
4. Undefined ( )

3.2. Partially meets requirement.

3.2.1. Current Language

3.2.1.1. Degree: 90%

MORAL requires that the number of dimensions, the range of subscript values for each dimension and the type of each array component be specified. The number of dimensions, the type and the lower subscript bound are determinable at compile time.

3.2.1.2. Language Description

An Arraytype has the form
array (Range) Simpletype
for a single dimension array or
array (Range,Range) Simpletype
for a two-dimensional array

Range takes the form
Integerconstant to Integerconstant or
Intererconstant : Intergerconstant

these being merely syntactic alternatives.

### 3.2.2. Modified/Extended Language.

MORAL could be extended to permit the upper bound of array subscripts to be a variable or expression whose value was determinable at the time of entry to the array allocation scope. This would require the addition of generic operators to make the actual bounds of arrays available within the source language.

1.0. Requirement: A7

The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

2.0. Language Elements.

2.1. Keywords.

group, endgroup, case, endcase, when.

2.2. Structures.

Groupdefinition, Fielddec, Casedec

2.3. Features.

N/A

2.4. Degree of Compliance.

1. Totally meets requirement ( )
2. Partially meets requirement (X)
3. Does not meet requirement ( )
4. Undefined ( )

3.2. Partially meets requirement

3.2.1. Current Language

3.2.1.1. Degree: 80%.

MORAL provides record types and permits them to have alternative structures. These alternative structures are fixed at compile time and fully discriminated at run-time.

#### 3.2.1.2. Language Description

A group in MORAL corresponds to a record in TINMAN. The fields of a group may be any Datadec, Proceduredec or Casedec. The latter form specifies a discriminated set of alternative structures that the group may take, dependent on the value of an integer or status field declared previously in the group.

#### 3.2.1.3. Limitations

The discrimination condition for the alternative structures defined in a casedec is restricted to a single field.

#### 3.2.1.4. Language/Requirement Conflict

The language constructions currently used to extract values from Casedecs make run-time checking of the current structure within Casedecs potentially expensive. This aspect of the language is currently under review.

### 3.2.2. Modified/Extended Language

This aspect of MORAL is currently under review. One possible outcome of this is that the current Casedec will be replaced by fully discriminated unions with access to the constituent values restricted to case-conformity statements along the lines of Algol 68.

1.0. Requirement: B1.

Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type of of a union type containing that type. Reference will retrieve the last assigned value.

2.0. Language Elements.

2.1. Keywords

assignment symbol: :=

2.2. Structures

Assignmentstatement, Expression, Variable.

2.3. Features.

N/A.

3.0. Degree of Compliance.

1. Totally meets requirement ( )
2. Partially meets requirement (X)
3. Does not meet requirement ( )
4. Undefined ( )

3.2. Partially meets requirement

3.2.1. Current Language.

3.2.1.1. Degree: 80%

Assignment is automatically defined for all types except Constanttypes (ie. those whose definition begins with the keyword const). Constanttype variables must have an initial value specified as part of their declaration, but may not thereafter be assigned to. Reference is automatically defined for variables of all types.

3.2.1.2. Language Description.

The Assignmentstatement is conventional. Similarly with reference to a variable, although the context will determine whether it is the value or the address of the variable which is required.

3.2.1.3. Limitations.

It is not possible to define assignment and access as part of encapsulated type definitions. However, it is possible to define these operations as procedures as part of the type definition.

3.2.1.4. Language/Requirement Conflict.

The language elements described in subsection 2 do not conflict with any HOL requirement.

3.2.2. Modified/Extended Language.

Extensions could be made to the language to permit the definition of assignment within type definitions. However such an extension

would raise the issue of inheritance of assignment (and similarly for other generic operations) for types derived from a type-'t' with user-defined assignment. These issues could become complex, for instance with union types including type 't'.

1.0. Requirement: B2.

The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

2.0. Language Elements.

2.1. Keywords

equality operator: =

2.2. Structures.

N/A

2.3. Features.

N/A

3.0. Degree of Compliance.

3.1. Totally meets requirement.

3.1.1. Language Description.

The operator "=" is automatically defined for all types, and may be used to compare data objects for identity.

3.1.2. Language/Requirement Conflict.

None.

1.0. Requirement: B3.

Relational operators will be automatically defined for numeric data and all types defined by enumeration.

2.0. Language Elements.

2.1. Keywords.

| | | |
|---|---|---|
| Comparators | = | (equals) |
| | < | (less than) |
| | > | (greater than) |
| | <= | (less than or equal) |
| | >= | (greater than or equal) |
| | <> | (not equal) |

2.2. Structures.

N/A

2.3. Features.

N/A

3.0. Degree of Compliance.

| | | |
|---|---|---|
| 1. | Totally meets requirement | ( ) |
| 2. | Partially meets requirement | (X) |
| 3. | Does not meet requirement | ( ) |
| 4. | Undefined | ( ) |

3.2. Partially meets requirement.

3.2.1. Current Language.

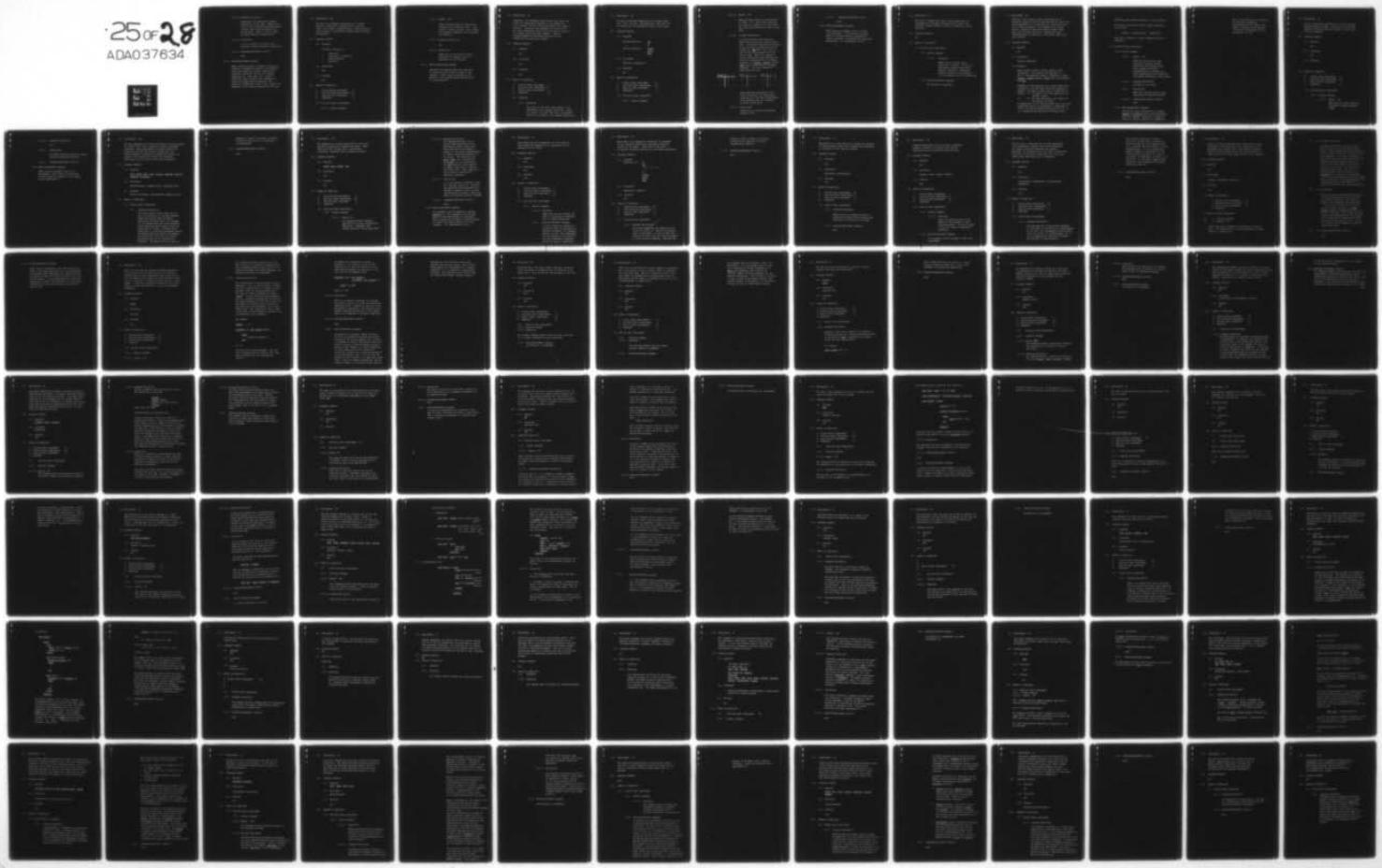
3.2.1.1. Degree: 95%

All six relational operators are defined for numeric data, and are automatically defined for types defined by enumeration.

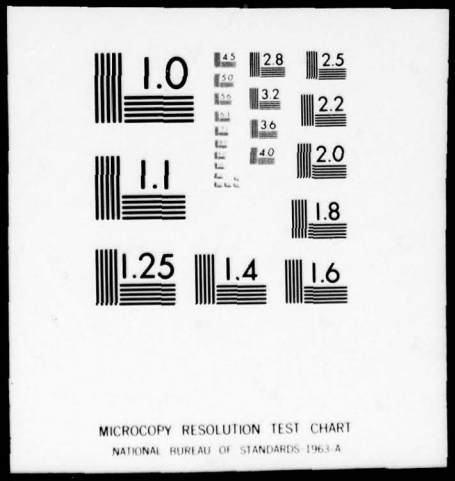
AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

25 OF 28
ADA037634



25 OF 28
ADA037634




1.0. Requirement: B10.

3.2.1.2. Language Description

Comparisons are permissable between expressions of any type. However only = and < > may be used between expressions of types other than the numeric and status types. (Status types in MORAL are types defined by enumeration).

3.2.1.3. Limitations

It is not possible to inhibit the ordering of types defined by enumeration.

3.2.1.4. Language/Requirement Conflict

None

3.2.2. Modified/Extended Language.

MORAL could be extended to permit the implicit ordering of enumeration types to be inhibited. Such types would then inherit equality and inequality operators, but not the ordering operators, eg. less than, etc. Further extensions could provide cyclic enumeration types and further generic operators (eg. successor, predecessor, first, last, ordinal etc) dependent on the ordering defined for the type.

1.0. Requirement: B4.

The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

2.0. Language Elements

2.1. Keywords

Arithmetic Operators:

+ (addition)
- (subtraction & negation)
* (multiplication)
/ (division)

2.2. Structures.

N/A

2.3. Features

N/A

3.0. Degree of Compliance

| | | |
|---|---|---|
| 1. | Totally meets requirement | ( ) |
| 2. | Partially meets requirement | (X) |
| 3. | Does not meet requirement | ( ) |
| 4. | Undefined | ( ) |

3.2. Partially meets requirement.

3.2.1. Present Language.

3.2.1.1. Degree: 90%

MORAL provides addition, subtraction, multiplication, division (with real result), integer division (with integer or fixedpoint arguments (and remainder) and negation.

3.2.1.2. Language description.

N/A

3.2.1.3. Limitations

MORAL does not provide an in-built operators for exponentiation or remainder for int division.

3.2.2. Modified/Extended Language.

Operators for exponentiation and remainder of integer Division could be added to the language. These could be provided by the addition of an operator definition facility. cf. Requirement E1.

1.0. Requirement: B5

Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

2.0. Language Elements.

2.1. Keywords.

N/A

2.2. Structures.

N/A

2.3. Features.

N/A

3.0. Degree of Compliance.

| | | |
|---|---|---|
| 1. | Totally meets requirement | ( ) |
| 2. | Partially meets requirement | ( ) |
| 3. | Does not meet requirement | ( ) |
| 4. | Undefined | (X) |

3.4. Undefined.

3.4.1. Reasoning.

The issue of truncation and rounding is not addressed by the language definition. There is, however, no reason why such a requirement should not be added to the language definition.

1.0. Requirement: B6

The built-in Boolean operations will include "and", "or", "not" and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode.

2.0. Language Elements.

2.1. Keywords.

Boolean operators: and
or

Logical operators: differ
union
mask

2.2. Structures.

Condition, Expression.

2.3. Features.

N/A

3.0. Degree of Compliance.

| | | |
|---|---|---|
| 1. | Totally meets requirement | ( ) |
| 2. | Partially meets requirement | (X) |
| 3. | Does not meet requirement | ( ) |
| 4. | Undefined | ( ) |

3.2. Partially meets requirement

3.2.1. Current Language

3.2.1.1. Degree: 70%

MORAL provides operators corresponding to "and", "or". When these operators are used in a Condition (ie. following if), they are evaluated in short circuit mode.

3.2.1.2. Language Description.

As described under requirement A2, MORAL does not provide a Boolean data-type. The Boolean expressions following an if are known as Conditions and may include the operators "and" and "or" both of which are evaluated in short circuit mode. MORAL also supports logical operations within Expressions on the bit representations of numeric operands (ie. integer, integer (Range), fixed (Size, Fractionbits) and byte operands). The truth tables for these operators are:

| differ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

| union | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| mask | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

These operators correspond to the logical functions "not equivalent", "inclusive or" and "and" respectively. These operators are not evaluated in short circuit mode.

3.2.1.3. Limitations.

MORAL does not provide the Boolean operator "not".

3.2.1.4. Language/Requirement Conflict

None

3.2.2. Modified/Extended Language.

MORAL could be extended to fully satisfy the requirement. This involves a certain amount of rationalisation as regards the overall treatment of booleans and conditional expressions. cf. requirement B6, B11.

1.0. Requirement: B7

The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

2.0. Language Elements.

N/A

3.0. Degree of Conformity.

3.3. Does not meet requirement

3.3.1. Current Language

3.3.1.1. Reasoning.

MORAL does not permit scalar operations on arrays. It does not distinguish between logical and physical structure. Assignment between records is only permitted if the variables are of the same type.

3.3.2. Modified/Extended Language.

No extension is proposed.

1.0. Requirement: B8

There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representations as characters, and between fixed point scale factors.

2.0. Language Elements.

2.1. Keywords

N/A

2.2. Structures.

Primary, Expression

2.3. Features

MORAL includes certain defined implicit type conversions. For instance the operators "+" and "-" are defined to apply to operands of types integer, integer (Range), fixed (Size, Fractionbits) and byte.

Operands of these operators are always (implicitly converted to the same type before the operator is applied. The type to which they are converted is either: (a) the type of the left operand if the context is weak.

or: (b) the type required by the context if the context is strong.

An Expression is in a strong context if it is on the right hand side of an assignment or comparison, if it is in an actual parameter position or if it is in a statement form requiring an Integer value.

Different type conversions apply to other operators.

The user can also force explicit type conversions by the rule:

Primary = Assignabletype : (Expression).

Note that a Primary is itself a degenerate form of Expression.

3.2. Partially meets requirement.

3.2.1. Current Language

3.2.1.1. Degree: 50%

MORAL provides explicit type conversions between integer, fixed point and floating point data, between the object, representation of integers and their representation as characters, and between fixed point scale factors.

3.2.1.2. Language Description.

See section 2.3. above.

3.2.1.3. Limitations.

MORAL does provide implicit type conversions in certain contexts.

3.2.1.4. Language/Requirement Conflict.

None

3.2.2. Modified/Extended Language.

The implicit type conversions eg. between integer and floating values could be removed from the language. Note that in MORAL the range of a numeric quantity is regarded as

part of the type and not a property of individual variables. Modification of this aspect would require revision of the handling of pointers (requirement D6) and parameters (C6, C8).

1.0. Requirement: B9

Explicit conversion operations will not be required between numerical ranges. There will be a run-time exception condition when any integer or fixed point value is truncated.

2.0. Language Elements.

2.1. Keywords.

N/A

2.2. Structures.

N/A

2.3. Features.

3.0. Degree of Compliance.

| | | |
|---|---|---|
| 1. | Totally meets requirement | ( ) |
| 2. | Partially meets requirement | (X) |
| 3. | Does not meet requirement | ( ) |
| 4. | Undefined | ( ) |

3.2. Partially meets requirement.

3.2.1. Current Language.

3.2.1.1. Degree: 50%

MORAL does not require explicit conversions between numerical ranges.

3.2.1.2. Language Description.

N/A

3.2.1.3. Limitations.

The MORAL definition does not discuss run-time exception conditions.

3.2.1.4. Language/Requirement Conflict.

3.2.2. Modified/Extended Language.

MORAL could be extended to meet this requirement. This would be a particular run-time exception condition to be handled as per requirement G7.

1.0. Requirement: B10.

The base language will provide operations allowing programs to interact with files, channels or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependant.

2.0. Language Elements.

2.1. Keywords.

join, leave, wait, stim, joinint, leaveint, waitint, controlq, interrupt.

2.2. Structures.

Mascotstatement, Typedefinition, Groupdefinition.

2.3. Features.

Virtual interrupts, Controlqueues, Channels, Pools.

3.0. Degree of Compliance.

3.1. Totally meets requirement.

3.1.1. Language Description.

Files and terminal input output are not built-in facilities of the base language. MORAL does however make the MASCOT operations available within the language via the syntax production Mascotstatement. These operations may be used to build up types such as "files", including their operations, in a machine independant manner. The MASCOT operations permit the sending and receiving of both data and control information and are not installation dependent. The MASCOT facilities may be

composed to enable the dynamic assignment of physical devices etc. as mentioned in the requirement.

3.1.2. Language/Requirement Conflict.

None.

1.0. Requirement: B11.

The language will provide operations on data types defined as powersets of enumeration types. These operations will include union, intersection, difference, complement and an element predicate.

2.0. Language Elements.

2.1. Keywords.

union, mask, differ, bits

2.2. Structures.

N/A.

2.3. Features.

N/A

3.0. Degree of Compliance

| | | |
|---|---|---|
| 1. | Totally meets requirement | ( ) |
| 2. | Partially meets requirement | (X) |
| 3. | Does not meet requirement | ( ) |
| 4. | Undefined | ( ) |

3.2. Partially meets requirement.

3.2.1. Current Language

3.2.1.1. Degree: 50%

The language provides powerset operations for union, intersection, difference. Complement and element predicate can be fabricated.

3.2.1.2. Language Description.

The language provides a set of logical operations which can be performed on numeric quantities. These operations are defined to operate on the machine representation of numeric variables regarded as a bit-vector. The operations are union, mask, (ie. intersection) and differ (ie. not equivalent). The bits operator may be applied to numeric quantities to extract individual bits. These operations may be used to fabricate and manipulate powersets.

3.2.1.3. Limitations.

It is not possible to define a type as a powerset explicitly. Moreover the operations mentioned above are defined to operate on the run-time representation of numeric quantities. These are obviously machine dependent.

3.2.1.4. Language/Requirement Conflict.

None.

3.2.2. Modified/Extended Language.

The language could be extended to include powerset as a type generator with suitable generic operations defined for powersets. This extension would require a rationalisation of boolean and logical quantities in the language. (cf. Requirements A2, B6)

1.0. Requirement: C1.

Side effects which are dependant on the evaluation order among the arguments of an expression will be evaluated left to right.

2.0. Language Elements.

2.1. Keywords.

N/A.

2.2. Structures.

N/A.

2.3. Features.

N/A.

3.0. Degree of Compliance.

| | | |
|---|---|---|
| 1. | Totally meets requirement | ( ) |
| 2. | Partially meets requirement | ( ) |
| 3. | Does not meet requirement | (X) |
| 4. | Undefined | ( ) |

3.3. Does not meet requirement.

3.3.1. Current Language.

3.3.1.1. Reasoning.

MORAL does not fully define the order of evaluation of operands. This is compiler-dependant.

3.3.2. Modified/Extended Language.

It would clearly be possible to define the order of evaluation of operands throughout the language. However, we feel that programming via side-effects is bad practice and that programming languages should not make it respectable.

1.0. Requirement: C2.

Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognised.

2.0. Language Elements.

2.1. Keywords.

operators etc:

<u>or</u>
<u>and</u>
<, <=, =, >=, >, <>
+, -
*, /
<u>differ</u>
<u>union</u>
<u>mask</u>

2.2. Structures.

Expression, Condition

2.3. Features.

N/A.

3.0. Degree of Compliance

1. Totally meets requirement (X)
2. Partially meets requirement ( )
3. Does not meet requirement ( )
4. Undefined. ( )

3.1. Totally meets requirement.

3.1.1. Language Description.

The various operators and comparators are listed in section 2.1. above in increasing order of precedence. It may be seen that there are 8 levels and that these correspond to their intuitive ordering. Brackets may

freely be used to enforce a particular association of operators with their operands where required.

3.1.2. Language/Requirement Conflict.

None.

1.0. Requirement: C3.

Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

2.0. Language Elements.

2.1. Keywords.

N/A.

2.2. Structures.

Expression, Loopstatement.

2.3. Features.

N/A.

3.0. Degree of Compliance.

| | | |
|---|---|---|
| 1. | Totally meets requirement | (X) |
| 2. | Partially meets requirement | ( ) |
| 3. | Does not meet requirement | ( ) |
| 4. | Undefined | ( ) |

3.1. Totally meets requirement

3.1.1. Language Description

MORAL allows the unrestricted use of expressions wherever both constants and references to variables are allowed.

3.1.2. Language/Requirement Conflict.

None.

1.0. Requirement: C4.

Constant Expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run-time.

2.0. Language Elements.

2.1. Keywords.

N/A.

2.2. Structures.

Casedec, Preset, Range, Constant.

2.3. Features.

N/A.

3.0. Degree of Compliance.

| | | |
|---|---|---|
| 1. | Totally meets requirement | ( ) |
| 2. | Partially meets requirement | ( ) |
| 3. | Does not meet requirement | (X) |
| 4. | Undefined | ( ) |

3.3. Does not meet requirement.

3.3.1. Current Language

3.3.1.1. Reasoning.

There are several points in the syntax of the language at which constants appear. In none of these is the full range of expressions permitted.

3.3.2. Modified/Extended Language.

The language could be extended to meet this requirement.

1.0. Requirement: C5.

There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types of exception handling, for parallel processes, for declarations or for built-in operators. There will be no special operations (eg. array sub-structuring) applicable only to parameters.

2.0. Language Elements.

2.1. Keywords.

N/A.

2.2. Structures.

Proceduredec, Parameterspec, Parameterpack, Expression.

2.3. Features.

N/A.

3.0. Degree of Compliance.

1. Totally meets requirement (X)
2. Partially meets requirement ( )
3. Does not meet requirement ( )
4. Undefined ( )

3.1. Totally meets requirement.

3.1.1. Language Description.

The name and type of each formal parameter of a procedure is specified in a Parameterspec. At the call, the corresponding actual parameters are written as a Parameterpack, ie. a list of Expressions, separated by commas and enclosed in round brackets.

The parameter mechanism in MORAL is basically 'call-by-value' although the value involved may be a reference-value (ie. a pointer) and give access to actual parameter data. It is as though each formal parameter is assigned its value from the actual parameter list and is thereafter an independant variable in its own right. In the case of const formal parameters this assignment is a dynamic presetting operation.

3.1.2. Language/Requirement Conflict.

None.

1.0. Requirement: C6

Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

2.0. Language Elements

2.1. Keywords

N/A

2.2. Structures

paratype, parampack, Expression

2.3. Features

N/A

3.0. Degree of Compliance

1. Totally meets requirement ( )
2. Partially meets requirement (X)
3. Does not meet requirement ( )
4. Undefined

3.2 Partially meets Requirement

3.2. I. Existing Language

3.2. I.I. Degree: 90%

Formal and actual parameters are required to agree in type. The dimensionality of array parameters is determinable at compile time.

### 3.2. I.2. Language Description

Procedure declarations include a specification of the type of each formal parameter. Actual parameters may be Expressions, but must yield a value of a suitable type for assignment to the formal parameter. In particular this will allow type conversion between the various numeric types, but will not allow the address of one type of record to be treated as the address of a different record type. The type of a formal parameter may be declared as a reference to an array of a certain type. In this case the dimensionality of the array must be specified: the corresponding actual parameter must yield the address of an array of the correct type and dimensionality.

### 3.2. I.3. Limitations

Arrays and records cannot be passed bodily (i.e. by copying their values) but must be passed by reference. The bounds of formal array parameters may be declared to be another parameter of the procedure provided that this is a const integer parameter appearing earlier in the list of parameters. This allows the array bounds to be communicated to the proceedure but requires the bounds to be passed explicitly at the point of call. It is not possible for the compiler to pass the bounds implicitly.

### 3.2. I.4. Language/Requirement Conflict

None

### 3.2. 2. Modified/Extended Language

MORAL could be extended to allow the bounds of an array to be passed implicitly with array parameters. As mentioned under Requirement A6, operators would be required to make the bounds available within the source language. The type constructors would need extensions to permit the definition of a type which represented an array whose bounds were unknown at compile time.

1.0. Requirement: C7

There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition there will be a formal parameter class for specifying the control action when exception conditions occur and a class for proceedure parameters.

2.0. Language Elements

2.1. Keywords

Const

2.2. Structures

Paratype

2.3. Features

N/A

3.0. Degree of Compliance

1. Totally meets requirement ( )
2. Partially meets requirement (X)
3. Does not meet requirement ( )
4. Undefined ( )

3.2. Partially meets Requirement

3.2.1. Existing Language

3.2.1.1. Degree 95%

The language includes parameter types for data which act as constants representing the actual parameter value at the time of call, those which rename the actual parameter, and parameters which may be procedures.

### 3.2.I.2. Language Description

The language has a single parameter passing mechanism. This is "call-by-value" although the value involved may be a reference value and give access to actual parameter data. The type of a formal parameter may begin "const": in this case the formal parameter acts as a constant representing the value of the actual parameter at the time of the call. The address of variables may be passed by declaring the formal parameter to be a reference type. This address may then be used in the body of the procedure to assign to the corresponding actual parameter.

For Example:

```
integer i  := 1;

procedure inc (Ref integer formal);

      begin
            [formal]:=[formal]+1
      end;

inc (i)
```

Following this program fragment, the variable i would contain the value 2. NB. The square brackets are the "dereference" operator.

Procedures may themselves be passed as parameters. For instance, given the declarations of "i" and "inc" above, the following program fragment is equivalent to the last line of the fragment above:

```
procedure apply (ref integer i,
                 procedure (ref integer) p);

    begin p(i) end;

apply (i, inc)
```

3.2.I.3. Limitations

There is no specific mechanism for defining the control action to be taken when exception conditions occur. However the ability to pass procedures as parameters, and especially procedures returning results can be used effectevely in the handling of exception conditions.

3.2.I.4. Language/Requirement Conflict

None

3.2.2. Modified/Extended Language

No extension is proposed. Where the possibility of an exception condition may be anticipated by the calling program (e.g. calling a procedure to access a device which may fault) then the ability to pass procedures as parameters is felt adequate to handle exceptions. In other cases such as store protection violations, overflow etc., where the exception is not anticipated, then it is felt that the response of the software is an important system issue. Within the MASCOT methodology, such an essential aspect of the system design would be

represented in the Network Diagram which precedes the detailed coding. Such an event would possibly be represented as a "Virtual interrupt", i.e. an external, assynchronous stimulus. The handling of such stimuli does not require extra MORAL language features.

I.O. Requirement: C8

Specification of the type, range, precision, dimension, scale and format of parameters will be optional on the formal side. None of them will be alterable at run time.

2.O. Language Elements

2.I. Keywords
N/A

2.2. Structures
N/A

2.3. Features
N/A

3.O Degree of Compliance

| | | |
|---|---|---|
| I. | Totally meets requirements | ( ) |
| 2. | Partially meets requirements | ( ) |
| 3. | Does not meet requirements | (X) |
| 4. | Undefined | ( ) |

3.3. Does not meet requirement
3.3.I. Current Language
3.3.I.I. Reasoning

The current language requires that the type, scale etc. of all formal parameters be fully specified.

3.3.2. Modified/Extended Language
No extension is recommended.

I.O. Requirement: C9

There will be provision for variable numbers of arguments, but in such cases all but a cnstant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

2.0. Language Elements

2.I. Keywords
N/A

2.2. Structures
N/A

2.3. Features
N/A

3.0. Degree of Compliance

| | | |
|---|---|---|
| I. | Totally meets requirements | ( ) |
| 2. | Partially meets requirements | ( ) |
| 3. | Does not meet requirements | (X) |
| 4. | Undefined | ( ) |

3.3. Does not meet requirement

3.3.I. Existing Language

3.3.I.I. Reasoning

The existing language does not support variable numbers of arguments.

3.3.2. Modified/Extended Language

The language could be extended to meet this requirement. The recommended extension would be to represent the formal parameter as on array type parameter, and to denote the (variable number of)array parameters via the Display construction. This solution is similar to that in Algol 68 and permits several sets variable numbers of parameters in a proceedure without the loss of compile-type checking.

1.0. Requirement: DI

The user will have the ability to associate constant values of any type with identifiers.

2.0. Language Elements

2.1. Keywords
const

2.2. Structures
Constant type

2.3. Features
N/A

3.0. Degree of Compliance

| | |
|---|---|
| 1. Totally meets requirement | (X) |
| 2. Partially meets requirement | ( ) |
| 3. Does not meet requirement | ( ) |
| 4. Undefined | ( ) |

3.1. Totally meets Requirement

3.1.1. Language Description

Constant values may be named as in a normal initialised declaration: the distinguishing feature is the keyword "const" appearing at the beginning of the type specification.

For Example:

```
const integer one: = 1;
```

Such a named constant may be used as a normal variable except of course that it may not be assigned to following the declaration.

### 3.I.2. Language/Requirement Conflict

None.

I.O. Requirement: D2

The language will provide a syntax and a consistent interpretation for literals of built-in data types. Numeric literals will have the same value (within the specified precision) in both programs and data (input or output)

2.0. Language Elements

2.I. Keywords
N/A

2.2. Structures
Number, string.

2.3. Features
N/A

3.0. Degree of Compliance

| | | |
|---|---|---|
| I. | Totally meets requirement | ( ) |
| 2. | Partially meets requirement | (X) |
| 3. | Does not meet requirement | ( ) |
| 4. | Undefined | ( ) |

3.2. Partially meets requirement

3.2.I. Existing Language

3.2.I.I. Degree: 60%
The language provides a syntax and consistent interpretation for literals of built-in data types.

3.3.I.2. Language Description
The language defines a syntax for literals of the types integer, fixed, floating and byte.

3.2.I.3. Limitations

Input/output is not defined in the language. The representation of literals of the basic types in data is therefore undefined.

3.2.I.4. Language/Requirement Conflict

None.

3.2.2. Modified/Extended Language

No extension is recommended.

1.0. Requirement: D3

The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initiatised at the time of their apparent allocation (i.e. at entry to their allocation scope.) There will be no default initial values.

2.0. Language Elements

2.1. Keywords
N/A

2.2. Structures
Presetoption, presetsequence, constant.

2.3. Features
N/A

3.0. Degree of Compliance

| | | |
|---|---|---|
| 1. | Totally meets requirement | (X) |
| 2. | Partially meets requirement | ( ) |
| 3. | Does not meet requirement | ( ) |
| 4. | Undefined | ( ) |

3.1. Totally meets requirement

3.1.1. Language Description
The declaration of individual variables includes a presetoption: if present, this specifies the initial value of the variable upon entry to the allocation scope; if not present, the initial value of the variable is undefined. In the case of simple variables, the value (if any) supplied by the presetoption must be a constant of the correct type: for compound variables, the presetoption must supply the correct number and type

of constant values corresponding to the elements of the compound variable.

3.I.2. Language/Requirement Conflict

The language elements described above conflict with requirements C4 (constant expressions) and C3 (expressions permitted) insofar as the initial value of the variables may not be the result of an expression. This restriction could however be removed without affecting other features of the language.

1.0. Requirement: D4

The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the oject code. Range and step size specifications will not be interpreted as defining new types.

2.0. Language Elements

2.1. Keywords
Integer, fixed, floating.

2.2. Structures
Simpletype.

2.3. Features
N/A

3.0. Degree of Compliance

1. Totally meets requirement ( )
2. Partially meets requirement (X)
3. Does not meet requirement ( )
4. Undefined

3.1. Partially meets requirement

3.1.1. Existing language

3.1.1.1. Degree: 80%
The language permits range specifications for individual integer and fixed point variables.

3.I.I.2. Language Description

The basic numeric types from which all others are derived are of the form:

integer
integer (Range)
fixed (Size, Fractionbits)
floating

Range takes the form:

Integerconstant to Integerconstant.

Sizeand fractionbits define the range for a fixed point variable: size is an integerconstant which defines the total number of bits in the machine representation of the variable (including the sign bit). Fractionbits is also an integer-constant and defines the number of those bits which are regarded as falling after the binary point: this number may be positive or negative and may be such that the binary point falls outside the stored representation of the value. The range specifications for fixed point variables are therefore limited to powers of two.

3.I.I.3. Limitations

There is no mechanism for specifying the range of floating point variables: their range and precision is assumed to be that of the single-precision floating point facilities of the execution environment.

Note also that range specifications for integer variables are optional; if not stated explicitly, the range of an integer variable is assumed to be that of the execution environment.

3.I.I.4. Language/Requirement Conflict

The range specifications described above conflict with requirement E7 as range specifications are construed as part of the type. However since this aids the security of use of pointers (cf. Requirement D6) and since explicit conversion operations are not required between different ranges, this is not felt to be a language defect.

3.I.2 Modified/Extended Language

The language could be extended to permit specification of range and precision of floating point variables, without affecting other aspects of the language.

1.0. Requirement: D5

The range of values which can be associated with a variable, array or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.

2.0. Language Elements

2.1. Keywords
N/A

2.2. Structures
N/A

2.3. Features

3.0. Degree of Compliance

3.2. Partially meets requirement (X)

3.2.1. Existing Language

3.2.1.1. Degree 95%

The range of values which can be associated with a variable, array or record component will be any built-in type or any defined type.

3.2.1.2. Language Description
There are no arbitrary restrictions on the structure of data: individual variables, the fields of structures and the elements of arrays may take values of any built-in or defined type.

3.2.I.3. Limitations

The range of values of an individual variable may not be defined to be a contiguous subsequence of an enumeration type.

3.2.I.4. Language/Requirement Conflict

None.

3.2.2. Modified/Extended Language

It would be straightforward to extend the language to permit individual variables to take values from a contiguous subsequence of (ordered) enumeration types.

1.0. Requirement: D6

The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other varables.

2.0. Language Elements

2.1. Keywords
ref

2.2. Structures
reference type

2.3. Features
N/A

3.0. Degree of Compliance

3.2. Partially meets requirement

3.2.1. Current Language

3.2.1.1. Degree: 95%

MORAL provides a pointer mechanism which can be used to build data with shared and/or recursive substructure. The MORAL pointer mechanism is as safe as any general scheme.

3.2.1.2. Language/requirement Description

Given any type "t", it is possible in MORAL to define a reference type by prefixing "t" with the keyword "ref". The values of a referncetype variable will be references to variables of type "t". Individual pointer variables are therefore restricted to refering to a single type of

value, although it is possible to define a pointer to variables of any desired type. The pointer property part of the type definition.

Pointers in MORAL are not restricted to composite data objects: this enables any values to appear in shared or recursive data structures.

Type definitions in MORAL may be prefixed with keyword const which indicates that values of that type may not be altered. This facility may be used to define pointer variables of the form:

Ref Constanttype.

Such a pointer variable may not be used to alter the variable to which it refers. This facility can be used to avoid some of the dangers in manipulating pointers.

3.2.I.3. Limitations

Pointers in MORAL are not prevented (by virtue of the language definition) from pointing to variables of a narrower allocation scope than the pointer itself. Given the ability to construct pointers to pointers, and the ability to use a pointer to alter the variable it refers to, it is not possible to ensure at compile time that pointers do not refer to variables of a narrower allocation scope. Note that a parameter passed by reference (as required under C7) acts as a pointer in this respect.

3.2.I.4. Language/requirement Conflict

None

### 3.2.2. Modified/Extended Language

No modifications or extensions are recommended.

1.0. Requirement: EI

The user of the language will be able to define new data types and operations within programs.

2.0. Language Elements

2.1. Keywords

type

2.2. Structures

Typedec, Usertype

2.3. Features

N/A

3.0. Degree of Compliance

| | | |
|---|---|---|
| 1. | Totally meets requirement | ( ) |
| 2. | Partially meets requirement | (X) |
| 3. | Does not meet requirement | ( ) |
| 4. | Undefined | ( ) |

3.1. Partially meets Requirement

3.1.1. Existing Language

3.1.1.1. Degree: 90%

The language allows the definition of new data types and the definition of new operations in the form of procedures.

3.1.1.2. Language Description

The new type can be defined as a Declarabletype, as a Proctype or as a Groupdefinition.

The examples below illustrate each alternative:

```
type card = array (1 to 80) byte;

type trigfunction = floating procedure (floating);

type complex = group

                floating Re, im;

                floating procedure modulus;

                   begin
                         answer Re*Re + im*im
                   end;

                endgroup;
```

Note that the last example includes the definition of an operation applicable to the type (procedure modulus).

3.2.I.3. Limitations

New operations can only be defined in the form of procedures; they may not be defined as infix operators.

3.2.I.4. Language/Requirement Conflict

None

3.2.2. Modified/Extended Language.

It would be possible to extend MORAL to allow for the definition of new operations in the form of infix operators. However the addition of such a capability raises complex issues of operator identification, inheritance

of generic operations, etc. No recommendation for the specific form this extension should take is therfore made.

I.O. Requirement: E2

The "use" of defined types will be indistinguishable from built-in types.

2.O. Language Elements

2.I. Keywords
N/A

2.2. Structures

2.3. Features

3.O. Degree of Compliance

I. Totally meets requirement (X)
2. Partially meets requirement
3. Does not meet requirement
4. Undefined

3.I. Totally meets requirement

3.I.I. Language Description

There are no semantic or syntactic differences or restrictions between the use of user-defined and built-in types.

3.I.2. Language/requirement Conflict

None.

1.0. Requirement: E3

Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

2.0. Language Element

2.1. Keywords
N/A

2.2. Structures
N/A

2.3. Features
N/A

3.0. Degree of Compliance

I. Totally meets requirement

3.I. Totally meets Requirement

3.I.I. Language Description

There are no default declarations.

3.I.2. Language/Requirement Conflict

None

1.0. Requirement: E4

The user will be able, within the source language, to extend existing operators to new data types.

2.0. Language Elements

2.1. Keywords
N/A

2.2. Structures
N/A

2.3. Features
N/A

3.0. Degree of Compliance

1. Totally meets requirement
2. Partially meets requirement
3. Does not meet requirement (X)
4. Undefined

3.3. Does not meet requirement

3.3.1. Current Language

3.3.1.1. Reasoning

The language provides no mechanism for the definition of operators. It is not possible either to extend the meaning of existing operators or to define new operators.

3.3.2. Modified/Extended Language

It would be possible to extend MORAL to permit the definition of infix operators and permit their extension to new operand types. However such an extension raises involved questions of operator identification, interpretation of generic operators, etc. No recommendation is therefore made as to the form this extension should take.

I.O. Requirement: E5

Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

2.O. Language Elements

2.I. Keywords
type,group,endgroup

2.2. Structures
Typedec, Groupdefinition

2.3. Features
N/A

3.O. Degree of Compliance

| | | |
|---|---|---|
| I. | Totally meets requirement | ( ) |
| 2. | Partially meets requirement | (X) |
| 3. | Does not meet requirement | ( ) |
| 4. | Undefined | ( ) |

3.2. Partially meets requirement

3.2.I. Existing Language

3.2.I.I. Degree: 95%

Type definitions permit the definition of both the class of data objects comprising the type and the set of operations applicable to the class.

3.2.I.2. Language Description

A type may be defined via a Groupdefinition. The Group definition may include both data fields which represent the values taken by variables of that type and procedures which define the operations applicable to varables of that type. Procedures defined in such a position are applicable only to variables of the defined type.

3.2.I.3. Limitations

(I) It should be noted that the lack of an operator definition facility in the language means that the operations applicable to a user-defined type can only be specified as procedures and not as in-fix operators.

(2) Where a Usertype has been defined using a built-in type as in:

type age = integer

then the userype is convertible into the base-type for the purposes of operations. However, the defined type can be prevented from inheriting the operations of the basetype as in:

type age = group integer age endgroup.

3.2.I.4. Language/Requirement Conflict

None

3.2.2. Modified/Extended Language

See under requirements EI and E4

1.0. Requirement: E6

The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian Products of existing types (i.e. as array and record classes), by discriminated union (i.e. as the union of disjoint types) and as the powerset of an enumeration type. These definitons will be processed entirely at compile time.

2.0. Language Elements

2.1. Keywords
type, group, endgroup, array, status, case, endcase

2.2. Structures
Usertype, Typedec, Casedec

2.3. Features
N/A

3.0. Degree of Compliance

3.2. Partially meets requirement

3.2.1. Existing Language

3.2.1.1. Degree: 80%

The language provides type definitions by enumeration and Catesian product. A form of discrimination union is also provided.

3.2.1.2. Language Description

The various forms of type definition can best be

illustrated by example:

I. enumeration

```
type suit = status (club, diamond, heart,
                                    spade);

type rank = status (two,three, four, five
                    six, seven, eight, nine,
                    ten, jack, queen, king,
                                        ace);
```

2. Cartesian product

```
type card = group
                rank rank;
                suit suit
            endgroup;

type pack = array (1:52) card;
```

3.0. Discriminated Union

```
type vehicle = group
                status (car,truck)vehicle
                                    type;
                case vehicletype
                when car: integer seating
                                    capacity
                when truck:floating load-
                                    capacity
                endcase

            endgroup
```

The example shows the form that union declarations take in MORAL. The discriminating field is explicitly and separatly defined (e.g. vehicle type above) and must be a status or integer (Range) variable. The Casedec then defines the alternative record structures and their dependencies on the values of the discriminatin field. Note that it is a many-to-one relationship.

```
e.g. group
          integer (1 to 10) tag;
          case tag
          when 1, 5 to 9: integer i,j,k
          when 2: floating f; integer e
          when other: byte b
          endcase
     endgroup.
```

The values of the discriminating field in different arms of the Casedec must obviously be disjoint.

3.2.I.3. Limitations

I. The language does not provide type definitions via powersets.

2. Although a Casedec includes a discriminating field, the construction does not guarantee consistent interpretation of data types. This is so because:

(a) the language provides explicit access to the discriminatin field independent of the subordinate fields. In particular, assignment to the

discriminating field is permitted independently of assignments to the subordinate fields.

(b) the language does not implicitly assign a suitable value to the discriminating field whenever an assignment is made to the subordinate fields. Note that many different values of the discriminating field could be associated with a given subordinate structure.

(c) the language does not demand that each access to the subordinate fields in a Casedec be accompanied by a check that the value of the discriminating field is consistent with the subordinate field structure.

3.2.1.4. Language/Requirement Conflict

The provision of run-time checks for consistent use of Casedecs may involve either a lengthy compilation process or some inefficiency in the object program. This conflicts with requirements J1 and L3.

3.2.2. Modified/Extended Language

(1) The language could be extended to provide type definitions by powersets of enumeration types. Such an extension would introduce some redundancies in the language and should be accompanied by a review of the capabilities for handling

boolean and logical quantities within the language. cf requirements A2,B6, BII.

(2) The existing language features for providing discriminated unions - the Casedecs etc. - are currently under review as they are not felt entirely satisfactory (cf. 3.2.I.4. above). One possible outcome is a replacement of these features by discriminated unions similar to Pascal or Algol 68.

1.0. Requirement: E7

Type definitions by free union (i.e. union of non-disjoint types) and subsettting are not desired.

2.0. Language Elements

2.1. Keywords
N/A

2.2. Structures
Casedec, Range.

2.3. Features
N/A

3.0. Degree of Compliance

3.1. Totally meets requirement

3.1.1. Language Description

The union facility is provided in MORAL via Casedec: this structure includes an explicit discriminant field.

New types may be defined via restricted ranges of integers, but in this case variables of the type are convertible into the base type for the purposes of operations. There is no functional difference between this approach and the alternative of ranges as a property of the variable rather than the type and disallowing inheritance of operations.

3.1.2. Language/Requirement Conflict

None

1.0. Requirement: E8

When defining a type, the user will be able to specify the initialisation and finalisation procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

2.0. Language Elements

2.1. Keywords
N/A

2.2. Structures
N/A

2.3. Features
N/A

3.0. Degree of Compliance

1.
2.
3. Does not meet requirement (X)
4.

3.3. Does not meet requirement

3.3.1. Current Language

3.3.1.1. Reasoning

Type definitions in the language do not allow the definition of operations to be automatically invoked when variables of that type are allocated and deallocated.

### 3.3.2. Modified/Extended Language

No extension is recommended

1.0. Requirement: F1

The language will allow the user to distinguish between scope of allocation and scope of access.

2.0. Language Elements

2.1. Keywords

lock, unlock, readonly, open

2.2. Structures

Locksequence, Keys, Groupdefinition

2.3. Features

Block structure

3.0. Degree of Compliance

| | | |
|---|---|---|
| 1. | Totally meets requirement | (X) |
| 2. | Partially meets requirement | ( ) |
| 3. | Does not meet requirement | ( ) |
| 4. | Undefined | ( ) |

3.1. Totally meets requirement

3.1.1. Language Description

MORAL is of conventional block structured form and follows the normal rules regarding scope of access to named variables etc. The language is designed to allow allocation to be performed at compile time for all variables except those declared inside recursive procedures (note that recursive procedures must be explicitly declared to be such).

In addition to the access restriction imposed by block-structuring, MORAL provides a system of "locks and keys" which may be used to create other protection structures. This is described under requirement F2.

### 3.1.2. Language/Requirement Conflict

None.

1.0. Requirement: F2

The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

2.0. Language Elements

2.1. Keywords
lock, group, const, readonly, unlock

2.2. Structures
Lockedsequence, Keylist

2.3. Features
N/A

3.0. Degree of Compliance

3.1. Totally meets requirement

3.1.1. Language Description

Definition of structures in MORAL is achieved via a Group definition. This defines the structure as a set of fields, each field being either data item or a procedure applicable to instsnces of the structure. Access to the fields can be limited via a system of "locks and keys". A set of fields enclosed between the brackets "lock" and "unlock" is known as a lockedsequence. Following the keyword "lock" is a set of identifiers which act as keys: only those instances of the group whose declaration includes a suitable key may access the fields of a Lockedsequence.

For Example:

```
type buffer =

    group
        lock
          array (0 to 10) message buffer;
          integer in, out
        unlock;

        lock reader,writer
          message procedure read;
          begin
            .
            .
            .
          end;

        lock writer;
          procedure write (message m);
          begin
            .
            .
            .
          end
        unlock

    endgroup;
```

The fields "buffer", "in" and "out" appear in a lockedsequence for which no keys are given: they are therfore accessible solely within the textual confines of the Groupdefinition, (e.g. to the procedures "read" and "write", but not where a buffer variable is declared). The procedure read appears in a Lockedsequence with two keys, "reader" and "writer": buffer variables declared with one of these keys may call this procedure. Thus, given:

```
buffer bI (reader), b2 (writer), b3;
```

then

```
bI. read, b2. write, b2. read
```

are all legal, but

```
bI. in, b2. in, b3. read, b3. write
```

are all illegal.

The example above is not particularly meaningful, but illustrates some of the capability of the scheme. Ther are other refinements such as the ability to define keys which give read-only access to fields: these are not described in detail here.

The scheme provides a flexible and simple mechanism for limiting access to details of implementation and for policing the use of interfaces. Note that the checking of access rights is carried out entirely at compile time. The scheme is integrated with other aspects of the lanuage (e.g. procedure parameters and pointer variables) to ensure that there are no loopholes in the access controls.

### 3.I.2. Language/Requirement Conflict

None.

1.0. Requirement: F3

The scope of identifiers will be wholly determined at compile time.

2.0. Language Elements

2.1. Keywords
N/A

2.2. Structures
N/A

2.3. Features
Block structuring

3.0. Degree of Compliance

1. Totally meets requirement (X)
2.
3.
4.

3.1. Totally meets requirement

3.1.1. Language Description

The language provides conventional block structured scope rules; the scope of identifiers is wholly determinable at compile time.

3.1.2. Language/Requirement Conflict

None.

1.0. Requirement: F4

A variety of application - oriented data and operations will be available in libraries and easily accessible in the language.

2.0. Language Elements

N/A

3.0. Degree of Compliance

Undefined.

3.4. Undefined

3.4.1. Reasoning.

The language definition does not specify any mechanism for defining or accessing library items. No language feature precludes the addition of library facilities.

1.0. Requirement: F5

Program components not defined within the current program and not in the bse language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other languages.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.4. Undefined

3.4.I. Reasoning

The language does not define any library mechanisms.

1.0. Requirement: F6

Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialised compools or libraries any user specified subset of which is immediately accessible from a given program.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.4. Undefined

3.4.I. Reasoning

The language does not define any library mechanisms.

1.0. Requirement: F7

The source language will contain standard machine independent interfaces to machine dependent capabiliies, including peripheral equipment and special hardware.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.4. Undefined

3.4.I. Reasoning

The language does not define any particular operating sustem, i/o or file system interfaces. However the MASCOT extension does provide a set of primitive operations which would allow such interfaces to be programmed in a machine independent way. See under requirements GI & G6.

I.O. Requirement: GI

The language will provide structured control mechanisms for sequential, conditioned, iterative and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

2.0. Language Element

2.1. Keywords

(statement seperator);
if, then, else, fi
case, when, endcase
for, while, do, endloop
recursive
join, leave, wait, stim, delay, joinint, leaveint
waitint, settransfer, suspend

2.2. Structures

Conditionalstatement, Casestatement, Loopstatement
Recursivity, Mascotstatement.

2.3. Features

N/A

3.0. Degree of Compliance

3.2. Partially meets requirement (X)

3.2.I. Current Language

3.2.I.I. Degree: 80%

The language provides control structures for sequential, conditional, iterative and recursive control. It also provides control structures for parallel processing and asynchronous interupt handling.

3.2.I.2. Language Description

Statements separated by semicolons are performed sequentially; conditional excution may be achieved via an if statement or by a case statement; iteration is achieved via a Loopstatement. Prodedures may be called recursively provided that they are suitable declared - viz. the keyword "recursive" appears in their declaration instead of "procedure". The MASCOT extension provides statements to control the execution of parallel processes and to handle asynchronous interupts.

3.2.I.3. Limitations

No control structure is present to handle exception conditions. The MORAL language is specifically designed to support the MASCOT methodology of constructing systems and exception handling is covered by this. For further discussion see under requirement G7.

3.2.I.4. Language/Requirement Conflict

None.

### 3.2.2. Modified/Extended Language

No extension is recommended. See under requirement G7.

1.0. Requirement: G2

The source language will provide a "go to" operation applicable to program labels within its most local scope of definition.

2.0. Language Elements

2.1. Keywords

goto

2.2. Structures

Label

2.3. Features

N/A

3.0. Degree of Compliance

3.2. Partially meets requirement
3.2.1. Current Language
3.2.1.1. Degree: 95%

The language provides a goto statement applicable to explicitly declared program labels.

3.2.1.2. Language Description

The language provides a "goto" statement of the form "goto label". The defining occurrence of the Label must be in the current or an enclosing block.

Note that Designational expressions, switches etc. are not provided.

3.2.I.3. Limitations

The goto statement may be used to cause a transfer of control to an enclosing block, i.e. to an 'earlier' scope level.

3.2.I.4. Language/Requirement Conflict

None.

3.2.2. Modified/Extended Language

The requirement could be fully satisfied by restrictions on the destination of goto statements.

I.O. Requirement: G3

The conditional control structure will be fully partitioned and will permit selection among alternative computations based on the balue of a Boolean expression, on the subtype of a value from a Discriminated union, or on a computed choice among labelled alternatives.

2.O. Language Elements

2.I. Keywords
if, then, else, fi
case, when, other, endcase

2.2. Structures
Conditionalstatement, Casestatement

2.3. Features
N/A

3.O. Degree of Compliance

3.I. Totally meets requirement

3.I.I. Language Description

The language provides an if - statement and a case - statement. The if statement is used to select alternative computations on the basis of a boolean expression. It takes the form:

if condition then statementsequence Elseoption fi

and is thus fully parlitioned. The Elseoption may be of the form:

else statementsequence

or may be omitted

The Case statement allows selection based on a computed choice among labelled alternatives.

case Expression Whenset endcase

and is thus fully partitioned. The Whenset is the set of labelled alternative computations. Each labelled alternative may take the form:

when Constant: Statementsequence

If the value of the Expression is equal to the Constant, then the statementsequence is evaluared. The Constant may be replaced by a range of specification, i.e. :-

Constant to Constant

with the obvious semantic, or by several Constants and Range specifications separated by commas. In this case, if the value of the Expression is equal to any of the constants or is in any of the ranges, then the statementsequence is executed. The last alternative of a Whenset may take the form

when other : Statementsequence

The same construction provides a selection on the basis of the subtype of a discriminated union, cf. requirements A7, E6 and E7.

3.1.2. Language/Requirement Conflict.

None

1.0. Requirement: G4

The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop and will not impose excessive overhead in clarity or run time execution costs for common special case termination conditions (eg. fixed number of iterations or elements of an array exhausted).

2.0. Language Elements

2.1. Keywords

for while from by to over endloop repeat, escape

2.2. Structures

Loopstatement, Controlpart, Body, Tail

2.3. Features

N/A

3.0. Degree of Compliance

3.1. Totally meets requirement

3.1.1. Language Description.

A Loopstatement is formed from a Controlpart, a Body and a Tail. The Body consists of a sequence of Statements (which may be empty) and is repeated according to the Controlpart. When the Loopstatement terminates according to the Controlpart, the sequence of statement forming the Tail is executed.

There are several alternative forms for the Controlpart. These allow iteration:

1. for each value from an enumeration type or integer range;
2. for each value from or variable in an array;
3. whilst a computed boolean expression remains true;
4. ad infinitum.

Where the Loopstatement has a control variable, e.g. one taking each successive value from an enumeration type, that variable is declared within the Controlpart and is local to the Loopstatement. The value of the control variable at loop termination may be passed out via assignment in the Tail to a variable global to the Loopstatement.

Entry to the Body of the loop may only be via the head of the loop. Two structure related jumps are provided which may only be used inside the Body of a Loopstatement: the repeat statement and the escape statement. The repeat statement causes the current iteration to be terminated: the Control part then determines whether there are any further iterations in the normal way. The escape statement causes execution of the entire Loopstatement to be terminated without execution of the Tail. Either of these statements may appear anywhere within the Body.

3.1.2. Language/Requirement Conflict.

None.

1.0. Requirement: G5

Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

2.0. Language Elements

2.1. Keywords

procedure recursive

2.2. Structures

Proceduredec, Recursivity

2.3. Features

N/A

3.0. Degree of Compliance

3.2. Partially meets requirement

3.2.1. Current Language

3.2.1.1. Degree : 95%

The language provides recursive as well as non-recursive routines.

3.2.1.2. Language Description

Recursive and mutually recursive procedures are fully supported provided they are explicitly declared to be recursive . This is achieved by replacing the keyword 'procedure' by the keyword 'recursive' in the Proceduredec.

1.0. Requirement: G6

The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

2.0. Language Elements

2.1. Keywords

join, leave, wait, stim,

2.2. Structures

Mascotstatement

2.3. Features

N/A

3.0. Degree of Compliance

3.2. Partially meets requirement

3.2.1. Current Language

3.2.1.1. Degree 80%

MORAL provides parallel processing primitives which allow a parallel process to gain exclusive use of resources for specified portions of their execution.

3.2.1.2. Language Description

The MORAL programming language is specifically designed to support the development of MASCOT-based programs.

The acronym MASCOT comes from 'Modular Approach to Software Construction, Operation and Test'. The MASCOT methodology covers all aspects of the Software effort from initial design to testing and operation of the target system.

MASCOT views the Software system as a network of activities connected by Intercommunication Data Areas. Activities execute in parallel, each taking the form of a single, sequential process. Synchronisation of the Activities occurs on their access to the Inter-communication data areas.

MORAL is designed for the coding of the individual activities and inter-communication data areas (IDA's) including the access procedures required for the latter.
(Access procedures are the means by which activities access the IDA's. They ensure the orderly flow of data through the IDA's by controlling the synchronisation of the calling Activities). The MASCOT scheduling primitives are therefore included in the MORAL language and take the form of operations on controlq variables, Controlq being a predefined type. These operations enable activites to gain exclusive access to resources for the periods of their execution.

The individual modules of the system (ie Activities and IDA's) which have been coded in MORAL are then integrated into the final system via MASCOT commands. The network of

activities then executes under the control of a MASCOT kernel which executes the MASCOT scheduling operations.

#### 3.2.1.3. Limitations

The network of Activities and IDA's which defines a MASCOT - based system is regarded as a static entity. ie. it is not possible for the running system to dynamically create a new parallel process. The ability to create new parallel processes exists as a MASCOT command rather than a MORAL facility.

### 3.2.2. Modified/Extended Language

No extension is recommended.

1.0. Requirement: G7

The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

2.0. Language Elements

None

3.0. Degree of Compliance

3.3. Does not meet requirement

3.3.1. Current Language

3.3.1.1. Reasoning

The language does not provide any mechanism to detect or respond to error conditions such as overflow, protection violations etc.

3.3.2. Modified/Extended Language.

No extension is recommended as MASCOT provides a design methodology for handling exception conditions. The ability of a software system to cater for overflow, store violations, hardware failures etc. is regarded as an important aspect of its specification and one that must be considered from the very beginning of its design. The MASCOT approach begins by constructing a Network Diagram which represents the flow of data between a set of cooperating sequential processes. Exception conditions would be represented in this Network Diagram along with other events such as data transfers from peripherals as "virtual interrupts", ie. asasynchronous external stimuli. Such "virtual interrupts" are

handled in the MORAL code via Mascot statements. These are discussed under Requirement G8.

1.0. Requirement: G8

There will be source language features which permit delay on any control path until some specified time or situation has occurred which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated like any other exception situation.

2.0. Language elements

2.1. Keywords

delay, wait, stim, joinint, leaveint, waitint, suspend

2.2. Structures

Mascotstatement

2.3. Features

N/A

3.0. Degree of Compliance

3.1. Totally meets requirement

3.1.1. Language Description

MORAL provides the MASCOT operations wait and stim which are used respectively to delay a given control path until a situation has occurred and to signal the occurrence of that situation. Additionally, the construction 'delay Expression' causes the execution of the issuing control path to be delayed until the time given by the Expression has elapsed.

A control path may set its own priority by the construction 'suspend Integerconstant' which causes the control path to be suspended, but to immediately become eligible for scheduling at the back of the schedulers list with a priority given by the Integer-constant.

Hardware interrupts are represented within MORAL by the predefined type interrupt. The following operations are applicable to interrupt variables:

1. joinint Variable, leaveint Variable these operations respecively request and release control of the interrupt control queue. They are used to guarantee mutual exclusion;

2. waitint Variable , may only be issued inside the dynamic brackets joinint and leaveint. It requests a transfer on the associated interrupt driven device, the control path being suspended while this transfer occurs.

3. settransfer Primary, Variable establishes the Data-address associated with the interrupt identified by the variable. The primary gives the address in the form of a Referencetype value. This operation may only be issued inside the dynamic brackets joinint and leaveint

### 3.1.2. Language/Requirement Conflict

None

1.0. Requirement: H1

The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

2.0. Language Elements

2.1. Keywords

N/A

2.2. Structures

N/A

2.3. Features

Phrase structured grammar

3.0. Degree of Conformity

3.1. Totally meets requirement

3.1.1. Language Description

The language is defined via a formal syntax which may be transformed into a one-track predictive form. It is therefore unambiguous and may be parsed without backtracking.
The source language is free format and follows the Algol family of languages. The semicolon is used as an explicit statement separator.
The keywords of the language are syntactically distinct from identifiers: the user is therefore free to choose mnemonic identifiers without constraint. Keywords and identifiers may not be abbreviated.

### 3.1.2. Language/Requirement Conflict

None

1.0. Requirement: H2

The user will not be able to modify the source syntax. Specifically, he will not be able to modify operator hierarchies, introduce new procedure rules, define new key word forms or define new infix operator procedences.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.1. Totally meets requirement

3.1.1. Language Description

The language has a fixed syntax. The parse of a given program is not modified by any declarations within the program.

3.1.2. Language/Requirement Conflict

None

1.0. Requirement: H3

The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

2.0. Language Elements

N/A

3.0. Degree of Compliance.

3.1. Totally meets requirement

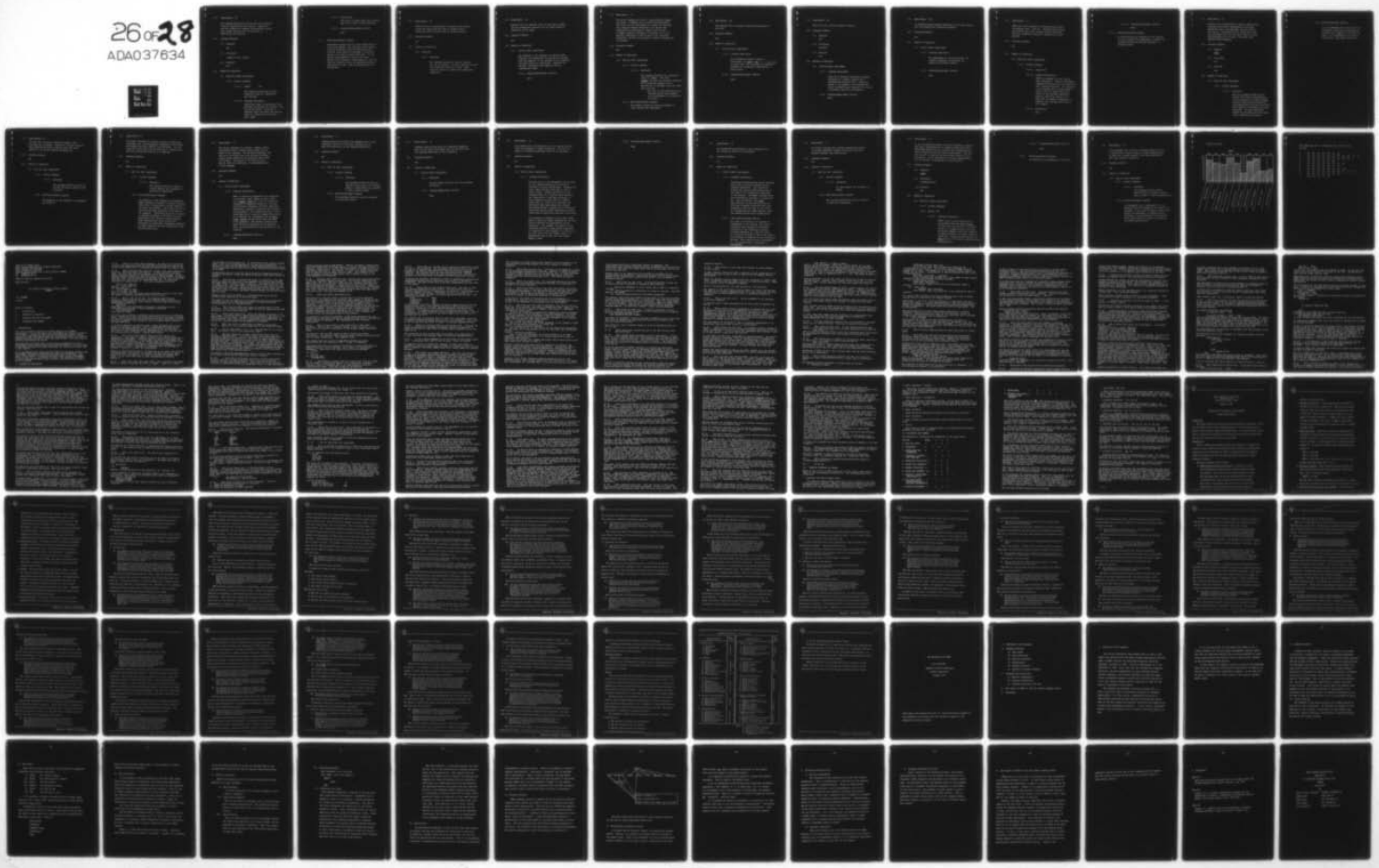
3.1.1. Language Description

The syntax of the language is defined in terms of terminal symbols such as commas, semicolons, keywords such as begin and identifiers, etc. Keywords are assumed to be syntactically distinct from identifiers: with a full character set this is achieved by the use of upper and lower case; where the character set is restricted, the keywords are distinguished by "stropping" (eg. enclosing them in primes).

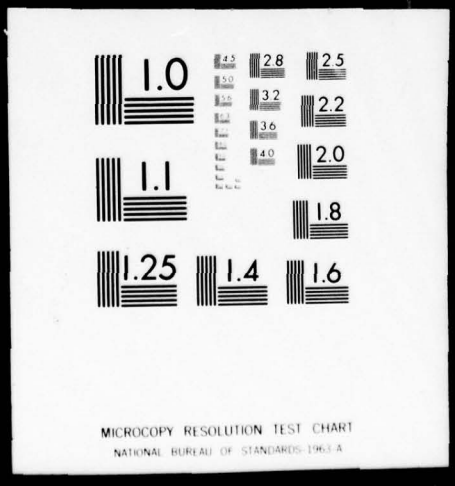
AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

26 OF 28
ADA037634



26 OF 28
ADA037634




1.0. Requirement: I4

The source language will permit the

1.0. Requirement: H4

The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

2.0. Language Elements

2.1. Keywords

N/A

2.2. Structures

Integers, Real, String

2.3. Features

N/A

3.0. Degree of Compliance

3.1. Partially meets requirement

3.1.1. Current Language

3.1.1.1. Degree 90%

The language definition provides formation rules for identifiers and literals.

3.1.1.2. Language Description

Identifiers take a conventional form. Literals are defined for integer, real and string values. Literals of compound types including user defined types are constructed from these basic types.

3.1.1.3. Limitations

The syntax of identifiers and literals does not include a break character.

3.1.1.4. Language/Requirement Conflict

None

3.1. . Modified/Extended Language

The present syntax does not allow identifiers to follow one another, and similarly with literals. Thus the formation rules for identifiers and literals could easily be extended to allow for spaces to be used as break characters. If the syntax were extended to provide for user-defined infix operators (requirements E1 and E4), then the formation rules for operators would be constrained to make them distinguishable from identifiers.

1.0. Requirement: H5

There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.4. Undefined

3.4.1. Reasoning

The language definition does not discuss the use of newline. It should however be noted that formation rules for lexical items could be chosen to satisfy this requirement completely.

1.0. Requirement: H6

Keywords will be reserved, will be very few in number, will be informative, and will not be usable where an identifier can be used.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.1. Totally meets requirement

The keywords of the language are syntactically distinct from identifiers by enclosure in primes. Thus the keywords of the language do not constrain the users choice of identifiers. Keywords are concise, informative and generally follow conventional usage.

3.1.2. Language/Requirement Conflict

None

1.0. Requirement: H7

The source language will have a single-uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.3. Does not meet requirement

3.3.1. Current language

3.3.1.1. Reasoning

The language provides two alternative forms of comment. The form:
comment any text not including semicolon;
may be inserted anywhere that a declaration or statement would be legal. Also the form:

; (Any text in which parentheses are matched) may be used anywhere as an alternative to a semicolon by itself.

3.3.2. Modified/Extended Language

The comment convention could be modified to fully satisfy the requirement.

1.0. Requirement: H8

The language will not permit unmatched parentheses of any kind.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.1. Totally meets requirement

3.1.1. Language Description

The language has several forms of bracketing, eg. begin...end , (...), if...fi. In every case, the occurrence of unmatched brackets in a program is regarded as a syntax error.

3.1.2. Language/Requirement Conflict

None.

1.0. Requirement: H9

There will be a uniform referent notation.

2.0. Language Elements

2.1. Keywords

N/A

2.2. Structures

Variable.

2.3. Features

N/A

3.0. Degree of Compliance

3.1. Totally meets requirement

3.1.1. Language Description

There is no syntactic distinction between reference to a simple variable and a parameterless function call. Similarly, access to an element of an n-dimensional array is syntactically identical to calling a procedure requiring n parameters.

3.1.2. Language/Requirement Conflict

None.

1.0. Requirement: H1O

No language defined symbols appearing in the same context will have essentially different meanings.

2.0. Language Elements.

N/A

3.0. Degree of Compliance

3.1. Totally meets requirement

3.1.1. Language Description

The requirement is fully satisfied, for example assignment and equality are denoted by different symbols.

3.1.2. Language/Requirement Conflict

None

1.0. Requirement: I1

There will be no defaults in programs which affect the program logic. That is, decisions which affect the program logic will be made either irrevocably when the language is defined or explicitly in each program.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.2. Partially meets requirement

3.2.1. Current Language

3.2.1.1. Degree 90%

3.2.1.2. Language Description

MORAL is intended to fully satisfy this requirement. However the present language definition does not explicitly state certain decisions. For instance, it is intended that a Casestatement should cover all cases that might arise yet the existing definition does not explicitly require this. Some work is therefore necessary to complete the language definition in this respect.

3.2.1.3. Limitations

None

#### 3.2.1.4. Language/Requirement Conflict

None

### 3.2.2. Modified/Extended Language

No modifications are required to the language to fully satisfy this requirement, however the language definition requires extension as discussed above.

1.0. Requirement: I2

Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

2.0. Language Elements

2.1. Keywords

dense

2.2. Structures

N/A

2.3. Features

N/A

3.0. Degree of Compliance

3.3. Does not meet requirement

3.3.1. Current Language

3.3.1.1. Reasoning

The only language element giving control over object representation of the program is the keyword dense. Structure definitions preceded by this keyword have an object representation given by a standard allocation algorithm. No other programmer control is available over the translators compilation strategies.

### 3.3.2. Modified/Extended Language

It is recommended that control over the translators compilation strategies be provided via 'pragmas' in the Algol 68 sense. No recommendation is made of the specific form these should take.

1.0. Requirements: I3

The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.3. Does not meet requirement

3.3.1. Current Language

3.3.1.1. Reasoning

The language definition does not include any form of compile time variables

3.3.2. Modified/Extended Language

No extension to the language is recommended. See under I4.

1.0. Requirement: I4

The source language will permit the use of conditional statements (eg. case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.3. Does not meet requirement

3.3.1. Current Language

3.3.1.1. Reasoning

The language definition does not include compile time variables or conditional compilation.

3.3.2. Modified/Extended Language

No extension to the language is recommended. MORAL provides the ability to define constants. These could, in particular, define aspects of the object environment. They could be held in libraries and made available to the user of the language. Translators would be at liberty to optimise programs on the basis of their knowledge of the values of these constants. These optimisations could include the evaluation of conditional statements dependent on these constants and the compilation of the selected paths only.

1.0. Requirement: I5

The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, or understandability of the language.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.1. Totally meets requirement

3.1.1. Language Description.

MORAL is explicitly designed to be extensible. It provides a small number of basic types (eg. integer, fixed etc.) and the facilities necessary to manipulate these, eg. variable declarations; assignment and basic operators; sequencing, iteration, selection and recursion. The remainder of the language provides a set of mechanisms which allow more complex structures to be defined, eg. mechanisms such as array, group, status and ref which allow new types to be generated, and procedure declarations which allow new operations to be generated. These various facilities form the base for the language.

3.1.2. Language/Requirement Conflict

None

1.0. Requirement: I6

Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.3. Does not meet requirement

3.3.1. Current Language

3.3.1.1. Reasoning

The language definition does not specify limits such as the maximum number of identifiers in a program or their maximum length.

3.3.2. Modified/Extended Language

The language definition could be extended to cover such limits.

1.0. Requirements: I7

Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

2.0. Language Elements

N/A

3.0 Degree of Compliance

3.1. Totally meets requirement

3.1.1. Reasoning

No such limits are built into the language definition

3.1.2. Language/Requirement Conflict

None

1.0. Requirement: J1

The language and its tranlators will not impose run-time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.1. Totally meets requirement

3.1.1. Language Description

Features such as file management are not part of the language definition but would be provided as library definitions. Programs not using these features would pay no cost on their account. Within the kernel of the language there is no feature which prevents the compilation of efficient programs, or which imposes run-time costs for unused generality, eg. there is no run-time cost associated with type definitions (except case declarations); storage can be allocated statically except within recursive procedures (which must be explicitly declared to be such).

The language is designed so that many common special cases are simply expressed by the user and are suitable for optimisation by the translator. For example, the syntax of the iterative construct allows direct expression of iteration for each element of an array, or for each value from an integer range or an enumeration type as well as other common special cases.

### 3.1.2. Language/Requirement Conflict

None

1.0. Requirement: J2

Any optimisations performed by the translator will not change the effect of the program.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.1. Totally meets requirements

3.1.1. Language Description.

Translators should produce object programs which have the effects defined by the language definition. The translation process may include optimiations made in the light of the translators knowledge of the program. The MORAL language makes much information available to translators eg. type information, the'const' qualification on data declarations etc. Also many language constructs have been designed so that the user is not forced to overspecify many common special cases and thereby prevent their optimisation.

3.1.2. Requirement/Language Conflict

The order of evaluation of operands of expressions is not defined: the order of evaluation may therefore be optimised according to the translators criteria. Programs dependent on a specific order of evaluation because of their side effect may therefore be affected by the translators actions. This conflicts with requirement C1. No modification or extension is recommended.

1.0. Requirement: J3

The source language will provide encapsulated access to machine dependent hardware facilities including machine language code instructions.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.3. Does not meet requirement

3.3.1. Current Language

3.3.1.1. Reasoning

No code inserts are available in MORAL.

3.3.2. Modified/Extended Language

The language definition could be extended to meet this requirement.

1.0. Requirement: J4

It will be possible within the source language to specify the object representation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the space/time tradeoff to the translator. If not specified, the object representation will be optimal as determined by the translator.

2.0. Language Elements

2.1. Keywords

dense

2.2. Structures

Groupdefinition

2.3. Features

N/A

3.0. Degree of Compliance

3.2. Partially meets requirement

3.2.1. Current Language

3.2.2. Degree 20%

3.2.1.2. Language Description

MORAL allows a group definition to be preceded with the keyword dense. In this case, the fields of the group (which is a composite data structure) are packed densely within the computer storage according to a known translator algorithm. Groups declared without the dense prefix do not have a defined object representation.

#### 3.2.1.4. Language/Requirement Conflict

None

### 3.2.2. Modified/Extended Language

No extension is recommended at this time.

1.0. Requirement: J5

The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.

2.0. Language Elements

N/A

3.0. Degree of Compliance

3.3. Does not meet requirement

3.3.1. Current Language

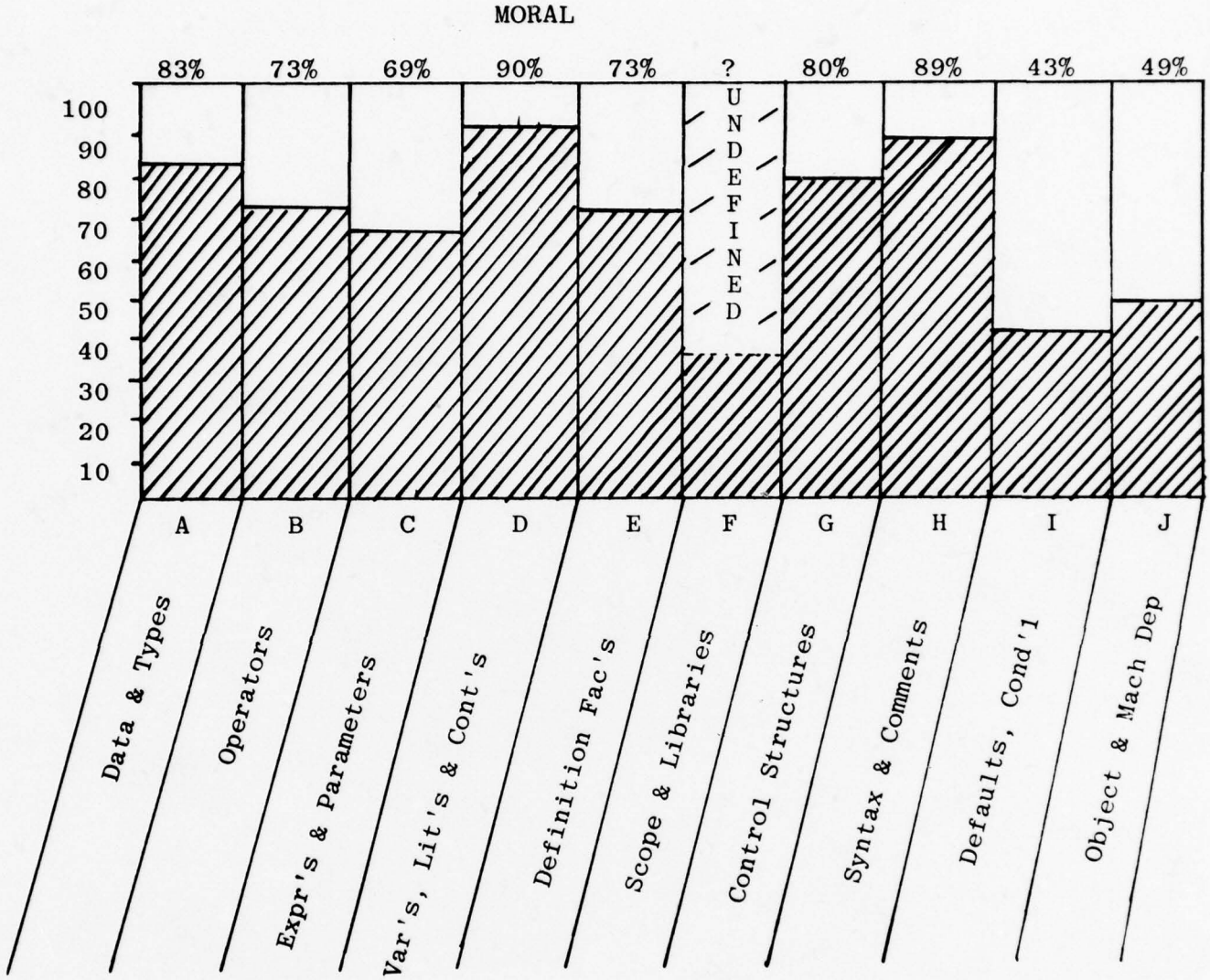
3.3.1.1. Reasoning

The language does not permit specification of whether routines have an open or closed implementation.

3.3.2. Modified/Extended Language

As recommended under requirement I2, the language could be extended by the addition of 'pragmas' in the Algol 68 sense. These would enable the programmer to communicate his knowledge of the properties of the program which would assist or guide the translator in making optimisations.

Evaluation Chart



The weightings used in producing this chart were as follows

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A | .26 | .25 | .09 | .09 | .06 | .13 | .12 | | | | |
| B | .16 | .12 | .09 | .10 | .08 | .09 | .03 | .06 | .05 | .15 | .07 |
| C | .06 | .15 | .16 | .14 | .16 | .15 | .09 | .03 | .06 | | |
| D | .16 | .16 | .16 | .08 | .20 | .24 | | | | | |
| E | .13 | .13 | .13 | .13 | .13 | .13 | .13 | .09 | | | |
| F | .10 | .14 | .13 | .20 | .20 | .10 | .13 | | | | |
| G | .14 | .08 | .14 | .14 | .08 | .14 | .14 | .14 | | | |
| H | .12 | .14 | .08 | .09 | .09 | .09 | .06 | .10 | .11 | .12 | |
| I | .16 | .13 | .14 | .14 | .16 | .14 | .13 | | | | |
| J | .22 | .25 | .19 | .18 | .16 | | | | | | |

(msg. # 16, 75033 chars)
Mail from USC-ISIC rcvd at 19-NOV-76 0643-PST
Date: 19 NOV 1976 0643-PST
From: SHIDELER at USC-ISIC
Subject: Barnes evaluation of RTL/2 against TINMAN
To: HOLWG at ISIC
cc: WHITAKER at ISI

Imperial Chemical Industries Limited.

Mond Division.

# An informal evaluation of RTL/2 against TINMAN

J G P BARNES

1-10-76

Table of Contents

1. Introduction

2. Comparative evaluation

3. Language features not needed

4. Discussion and summary

## I INTRODUCTION

This is an informal evaluation of RTL/2 against the "TINMAN" requirements. In certain areas the requirements go outside the domaiin of the RTL/2 language and more concern the operating system. Where appropriate such requirements have been considered against the range of operating systems written in RTL/2 and known as MTS. This particularly concerns requirements G6 and G8.

This evaluation has been done on a voluntary unfunded basis and is consequently less complete than might have been desired. In some cases it has been necessary to propose extensions in outline only rather then in full detail.

As far as possible the author has performed the evaluation assuming that the TINMAN requirements themselves are completely acceptable. The author does, however, have misgivings about some of the requirements. These are described in a separate document 'Some comments on TINMAN'. In some areas this document helps to explain the differences between RTL/2 and TINMAN.

## II COMPARATIVE EVALUATION

A1 (T) RTL/2 is a fully typed language. The type of all variables is fully and explicitly given in their declaration and the type of the result of all operations is unambiguously determiinable at compile time.

A2 (P) RTL/2 provides data types for integer, real (also fixed point fraction), byte and arrays and records. It does not have a Boolean type. The byte type can be considered as the character type. If the Boolean type were added then one would have to decide whether the dyadic operators LAND, LOR should extend to Booleans with the risk of confusion with the short circuit control operations AND, OR. It would seem reasonable to extend them. It would then be possible to write

```
     BOOL B:=X>Y LAND Y>Z;
```
but not `B:=X>Y AND Y>Z;`
whereas either of
```
     IF X>Y LAND Y>Z THEN
```
and `IF X>Y AND  Y>Z THEN`
would be allowed with the latter being evaluated in short circuit mode. The impact on existing implementations would be small.

A3 (F) This is not met at all. The language specification explicitly states that the precision of floating point arithmetic is implementation dependent. Such a feature could be added by allowing declarations of the form
```
     REAL(10)R;
```
specifying R to have 10 decimal (or binary?) significant digits. A scope definition allowed in block headings of the form
```
     REAL PRECISION 10;
```
would also be needed.

Of course real variables of different precisions would not be different types. Reference to real variables could not have a precision attached and the use of references and procedures would allow the import of lower precision arithmetic into scopes for which a general higher precision had been specified.

It would be expensive to add more than a token implementation of this feature to existing compilers. Such an implementation would merely check that the specification given can be met by the one and only precision of reals supported by the implementation; if it did not then the program would not compile. Better than nothing.

A4 (P) The fraction data type of RTL/2 partially meets this requirement. The range is, however, always (-1,+1) and the precision is implementation dependent. Fractions meet the general requirement for fixed point approximations to real numbers when the object machine does not have floating point hardware. To add range and stepsize specification to RTL/2 plus compile time scale factor management would take us back to RTL/1. Feasible but expensive and I do not wish to propose an extension to meet this requirement.

However to add precision only in the same way as for reals (A3) would be straightforward. The duality of integers and fractions within RTL/2 would make this also apply to the range of integers throughout the scope. As for reals more than a token implementation would be expensive.

A5 (P) RTL/2 has a byte data type rather than a character data type. Bytes are merely a subrange (0..255) of integer. There is a literal

representation of characters 'A', 'B' which gives their unique internal value. This value is unalterable at tun time and is explicitly specified in the language definition. RTL/2 generally meets the spirit of the requirement but it does not allow new enumeration character types to be added.

To add such a feature given the use of bytes for characters would be extremely difficult. A genuine separate character type would really be needed.

A6 (P) The user has to specify the number of dimensions, the upper subscript bound and the type of the elements in the array declaration. The lower bound is always 1 and is implicit. The upper bound is fixed at compile time. The requirement is therefore not fully met since the lower bound can only be 1. Since all arrays are global within a module the upper bound is formally determinable at entry to the scope but clearly the spirit of the requirement demands dynamic arrays which RTL/2 does not have.

Dynamic arrays could be added in a straightforward way by merely allowing their declaration in any block head.

The impact of such an addition on the compilers would be considerable and mean that the run time stack management routines would need reconsideration for all implementations.

A7 (F) RTL/2 does not meet this since all records have a fixed structure. Such a feature could be added to RTL/2 by expanding the MODE definition in a tree structured way.

B1 (T) RTL/2 fully meets this. Users can declare variables of all data types - including control types such as procedures and stacks - and assign and refer to these. Access to array and record components is also possible. RTL/2 does not have encapsulated type definitions (E5) and so user defined access and assignment is irrelevant.

B2 (T) RTL/2 has built in operations to compare any two data objects. These are = for primitive types and :=: for reference types.

B3 (T) The six operations = # >= > <= < are defined for all numeric data. See answer to E6 for new enumeration types.

B4 (P) The only operation of those listed which RTL/2 does not provide is exponentiation. It could be added, (with some misgivings) relatively simply by an operator**. The range of types allowed as operands needs some thought particularly with the added complication of the fraction type. A possibility would be to allow the first operand to be REAL or FRAC and the second INT or REAL. The result would have the same type as the first operand. Its precedence would need to be above that of the multiply and divide operators. It would probably be best to keep the shift operators as the highest precedence and so this would mean making them 7 and exponentiation 6.

The impact on existing implementations would be fairly small although it would entail the writing of additional control routines.

B5 (T?) I think RTL/2 fully meets the spirit of this requirement. Certainly all out of range conditions lead to overflow and never truncate the most significant digits. (The detection of the overflow condition is specified as machine independent and so on some machines an

illegal program could go undetected). However, automatic rounding does occur when converting from fine integer to integer and fine fraction to fraction. Maybe this is outside the requirements of TINMAN as being an unnecessary elaboration. If, however, it were to be stated explicitly then an operator such as TRUNC could be added. See comments on A4 - with hindsight truncation seems better than rounding in view of the range assymmetry of fractions.

B6 (P) Although RTL/2 does not have a Boolean data type nevertheless it has the necessary logical operations. The logical operations are LAND (and), LOR (or), NEV (Nor), NOT acting on integers and bytes (not NOT on bytes by an oversight). There are also short circuit control operations AND and OR acting on comparisons in conditions. Clearly the full requirements of TINMAN can only be sensibly met by adding a Boolean data type. Extending the existing operations to meet that type would then be trivial. See also the comments on A2.

B7 (F) RTL/2 has scalar operations only.

Syntactically it would be easy to extend RTL/2 to allow aggregate operations. The semantics are difficult with respect to arrays of more than one dimension since there would be confusion between operating on the first (or any) level of references and the ultimate elements themselves. The spirit of RTL/2 would suggest operating at the top level only whereas the typical matrix user would need to operate at the ultimate elements. Clearly if operators like + between arrays were allowed then these could only operate at the ultimate elements. For equality comparison one could distinguish by :=: and = between the extreme levels but assignment poses a real problem.

In conclusion it would be reasonable to add these operations for one dimensional arrays only although the wisdom of doing this in an Algol-68 based language is debatable. The implementation cost would be significant.

B8 (P) RTL/2 allows implicit type conversions in the 'safe' direction. That is byte > integer, integer > real, fraction > real. There are also implicit conversions for the double length fixed point forms.

The operators INT and REAL could be used to denote the former conversions. The operator TRUNC could be used to denote conversion from fine to normal fixed point. See comments on B5.

This change would be trivial to implement although its value, particularly for the conversion byte > integer is debatable.

Requirement B9 covers the conversion from big to normal form where an explicit operator is not required. This leads to an unfortunate assymmetry. Given INT I,J,K; FRAC F,G,H; we are now allowed

```
     I:=J*K;
```

but not

```
     F:=G*H;
```

for which we need

```
     F:=TRUNC(G*H);
```

B9 (T) RTL/2 does not require explicit conversions from the ranges big to normal of fixed point values. If a fixed point (integer or fraction) value is truncated then a run time exception condition (overflow will occur. This may or may not be detected.

B10 (T) RTL/2 does not include inbuilt operations specifically aimed at file or I/Ø processing. The philosophy has been to write a set of procedures to perform such operations using the ordinary language features. The features which are used significantly for language features. The features which are used significantly for I/Ø and error recovery are SVC data areas, procedure variables, bytes and strings.

In particular procedure variables are used for dynamically assigning and reassigning devices and for specifying user control of exception conditions. SVC data allows this to be done independently by each task in the system.

B11 (P) RTL/2 does not have general power sets. Nor does it have user defined enumeration types as such. It does however have the commonly required ability to manipulate bit patterns. Such patterns are held in integers (or bytes). Thus power set up to cardinality 16 (minimum word length allowed) are easily manipulated by the user. The operations available are

| | | |
|---|---|---|
| union | : | LAND |
| intersection | : | LOR |
| difference | : | NEV |
| complement | : | NOT |

If the bottom n bits of an integer are used to denote the set with the remaining bits set to zero then the operations NEV and NOT strictly should be followed by a further masking operation to preserve the convention. If, however, the top bits are ignored then this is not necessary. An element predicate can be synthesised using LAND. Thus if X represents an element and Y a set then X LAND Y will be non zero if and only if X is in Y.

The simplest way of extending RTL/2 to meet this requirement (and that of general enumeration types) would be via a pre-processor. This would be fairly easy and would have no impact on existing implementations.

C1 (P) Almost by oversight RTL/2 is not perfect here. Although the order of evaluation of procedure parameters and array subscripts is specified as left to right that of ordinary dyadic operands is not but ought to be.

The correction of this so that RTL/2 fully meets this requirement is trivial and would have neglible impact on existing implementations.

C2 (P) It is a value judgement as to the extent to which RTL/2 meets this. I personally feel that it has too many. X/Y/Z is allowed.

It would be trivial to change the language to reduce the number of precedence levels and fairly easy to insist on brackets in cases such as X/Y/Z. However, the former change would have a nasty impact on existing programs.

C3 (P) RTL/2 meets this almost perfectly. The only true exception is that full procedure expressions are not allowed to stand for the procedure in procedure calls. A procedure call must use a literal procedure (constant of type) or procedure variable (reference to type) explicitly. This restriction was imposed to improve readability. There is a similar restriction preventing the subscripting of general array expressions and component selection of general record expressions again for the sake of readability but perhaps this topic is outside the scope of the requirement and so does not conflict.

The relaxation of these restrictions requires a trivial change to the language definition and probably only minor changes to existing compilers.

C4 (F) RTL/2 does not meet this. Constants can be named and so many of the maintenance problems are overcome. However, it would be useful to allow constant expressions as array lengths, etc. A recent proposed extension to RTL/2 permits this for integer expressions involving integer operators and evaluated modulo 2'5. This would be easy to implement.

C5 (T) RTL/2 fully meets this. The parameter mechanism is simply assignment and any expression allowed on the RHS of an assignment is allowed as an actual parameter.

C6 (T) RTL/2 fully meets this in the sense that formal and actual parameters always agree to the same extent as the left and right sides of assignment statements. Any laxness in RTL/2 is reflected by the equivalence of the parameter mechanism to assignment and would be removed by changes to assignment demanded by requiement B8.

In particular the number of dimensions for array parameters is determined at compile time but the length of array parameters is passed as part of the parameter in accordance with the requirements.

C7 (P) RTL/2 has only one class of parameter, the call by value of Algol 60. This provides, with data of appropriate type, almost all the functions of the four classes of TINMAN. Thus Class 1 - constants representing actual value.

This is the normal case obtained for example by a formal parameter of mode INT with an actual parameter being any expression. Of course the formal parameter can be assigned to and this conflicts with the requirement but on the other hand the actual parameter is not damaged. (RTL/2 has no read-only data types anyway).

Class 2 - rename the actual parameter.

This is the case where the formal parameter is for example of mode REF INT and the actual parameter is of mode INT.

Class 3 - exception control handling.

This is catered for in a primitive way by using formal parameters of mode LABEL (for unrecoverable errors) or mode PROC (for recoverable errors).

Class 4 - procedure parameters.

This is just the case where the formal parameter is of mode procedure. The actual parameter is of course restricted to have an identical specification.

Given that one accepts this TINMAN requirement RTL/2 would clearly need some significant changes to meet it.

Firstly a read-only attribute would be needed for all data if the present simple assignment rule were maintained. On the other hand a simple constraint that all formal parameters be read-only (as for statement control variables are) would be simple to implement although might bear heavily on existing programs. It would be more secure but marginally less efficient in some cases where additional copying would then be necessary.

Secondly a more formal exception handling mechanism akin to (but hopefully better than) the ON condition of PL/1 would have to be added as a replacement for the LABEL mode. This could be done but would be a

large change entailing a significant amount of redesign. The requirement that exception handling control parameters will be specified on the call side only when needed means that a pre processor can not be used to provide this.

Thirdly there is the implication that procedure variables are not allowed except as parameters. This would be easy to do but would make a mockery of our input-output standards and entail a lot of change to existing programs.

C8 (F) RTL/2 does not meet this. It would be possible to make the following optional or general in procedure specifications.

i) Specifying INTORFRAC to cover either INT or FRAC in a specific instance.
ii) Omitting details of references to arrays or records - just REF ARRAY or REF RECORD.
iii) Omitting specifications of procedure parameters.

The compiler would then only compile the procedure if this could be done without using absent information. It would also check that at procedure call the actual parameters were self consistent. Case (ii) above would be the most valuable. This would be a relatively expensive change since it would also involve changing the linkage checks across separately compiled modules. It would not, however, affect existing programs.

C9 (F) RTL/2 does not meet this. A possible extension to do so might be to allow the final parameter to look like an array in disguise. Or use a different keyword as in

```
PROC MAX(MULTIPLE REAL X)REAL;
```

The number of parameters could be obtained by LENGTH X as for arrays and they could be accessed by subscripts with security provided by the existing array bound checks.

This would be a fairly awkward change to existing implementations but not too bad.

D1 (T) RTL/2 meets this through the use of the LET facility thus

```
LET PI=3.14159;
```

D2 (P) RTL/2 almost meets the spirit of this requirement. Firstly there is a well defined syntax and interpretation for the numeric types byte, integer, fraction and real. (The language specification manual is a bit untidy in this area but the intention is clear). Any misgivings regarding the precision of constants would be cleared by the proposals in A3 and A4.

As far as equivalence between constants in programs and data RTL/2 rather avoids this question by not defining input-output in the language itself. We can, however, compare the standard I/Ø package with the language. Reals are OK but fractions are a problem. For example the language will not accept 1.OBO as legal (too big) but the input routines interpret this as the maximum allowed value and accept it. On the other hand every value output can be reinput. Some work needs doing in this area.

Perhaps it should be pointed out that existing RTL/2 compilers behave well in this area because they handle all program constants in byte arrays and make no attempt to utilize the real variables of the

compiling machine.

D3 (P) RTL/2 fails to fully meet this because it allows default initial values.

It does, however, permit the user to specify initial values as part of declarations both in static data in data bricks and dynamic data on block entry.

There is a default initial value of zero for variables of numeric mode (byte, integer:0; fraction:0.0B0; real:0.0) in data bricks only. There is no default value for dynamic variables.

There is no provision for run time testing for initialisation.

The removal of the zero default initial value is trivial (and welcome). The addition to existing implementations of a user option for run time testing would be rather expensive since it intimately affects all compilers.

D4 (F) This is not met at all. See the comments on A4 regarding fixed point stepsize.

To add more than a token implementation of this would be expensive as for A3.

D5 (P) RTL/2 allows arrays as components of records and vice versa but does not allow arrays to be components of arrays nor records as components of records. This simplification was partly to ease implementation and partly to ease training and increase credibility in a sceptical environment. The relaxation of these restrictions requires only a small change to the language definition but it would be rather expensive to change existing implementations.

The requirement for variables to take a contiguous subsequence of an enumeration type would best be met by a preprocessor.

D6 (P) RTL/2 allows too much here. It provides a pointer mechanism (REF) as required but allows it to extend to simple variables as well as arrays and records. It is currently not completely safe although I know of no case where a scope error has occured.

To alter RTL/2 to meet this requirement is straightforward. One merely deletes REF scalar variables. They would however still be needed for formal parameters of the second class. The mechanism of dereferencing would need redescribing and the left hand operator VAL could be removed. This would all be quite straightforward to implement but would have a large impact on existing programs. It would however give the total security required.

However the implications of some of the other changes (e.g. A6, dynamic arrays) would mean additional complication to ensure continued security. This would be more difficult.

E1 (P) RTL/2 includes user defined MODEs (records) and so has the basic ability to define new data types. It cannot, however, define new infix operators. An extension to do this might be as follows. Define a new operator as a procedure with two parameters and result with name giving the denotation of the operator and its precedence. E.g. the procedure might read

```
PROC INFIX(1) ++ (REAL X,Y)REAL;
```

where 1 is the precedence. Note that I would prefer all new user defined operators to have the same precedence and so the precedence value need not be stated. For unambiguous syntax (see H4) it is essential that the name of an operator be lexically distinct from other identifiers. Thus it should be composed of characters such as those traditionally used for operators e.g. + - / etc. The model of POP-2 is suggested.

Such an extension, on its own, would be fairly easy to add to existing implementations. However, other aspects of RTL/2 records (e.g. all static) render infixed operations on them less useful than might be expected.

E2 (F) The user defined types of RTL/2 (record classes) are used in a way distinct from scalar types and array types. I suspect that such user defined types are not what TINMAN has in mind and therefore that the answer to the previous question is optimistic. If TINMAN does not consider record types as defined types then RTL/2 does not have defined types and this question does not apply.

If on the other hand record types are defined types then RTL/2 fails to meet the requirement because, for example, it does not allow local records.

Local records could be added by merely allowing their declaration in any block head. This would add considerable complication to the compilers largely to cater for the case where the record has components which are arrays.

If other types were added (see E6) probably via a preprocessor then since they would map onto existing types their use would be indistinguishable in the sense intended.

E3 (T) All data types and variables have to be defined or declared explicitly in an RTL/2 module. There are no default attributes.

E4 (F) RTL/2 does not meet this. If user defined operators were added as in E1 then they would allow existing operators as the name e.g.

```
PROC INFIX + (.....)
```

Uses of existing operators which invoke their extended definitions would be implemented as procedure calls and commutativity would not be assumed.

E5 (T) RTL/2 meets this in respect of its existing types, explicitly or implicitly. For example a record class defined by

```
MODE P(INT X,Y)
```

indicates the class of objects comprising the type, the constructor P, the selectors X and Y. Other operations are defined by procedure declarations and P does not inherit any of the operations on integers.

Extensions to RTL/2 to give other defined types (see E6) would similarly satisfy this requirement.

E6 (P) RTL/2 allows Cartesian products but does not allow new enumeration types, discriminated unions or power sets of enumeration types.

The language could be extended to include these as follows

i) enumeration types

TYPE name (literal name list)
would define 'name' as an enumeration type with the literalnames as in brackets. The optional keyword ORDERED would be allowed before TYPE. See comment B3 in the document 'Some comments on TINMAN' by the author. Subsequences of ordered enumeration types could be expressed thus

TYPE name (firstname .. lastname)
where firstname and lastname are literal names in some ordered enumeration type and firstname preceeds lastname.

ii) discriminated unions
UNION name (typenamelist)
would define 'name' as a union of the disjoint types in brackets.

iii) power sets
POWERSET name (typename)
would define 'name' as a powerset of the enumeration type in brackets.

For clarity and consistency the keyword MODE currently used to define records would be replaced by the keyword RECORD.

Extensions (i) and (iii) could be handled by a preprocessor and need have little impact on existing implementations. The new types would be mapped onto integers or bytes and there would therefore be constraints on their cardinality. The usual operations would all map easily. See B2, B3 and B11.

The discriminated union would need massive extensions to existing compilers. Notation similar to Algol 68 could be used. Access to values of a discriminated union could be via references and 'a conforms to and assign' operator (::=) in conditions.

IF X::=Y THEN
would assign the value in the discriminated union (variable or reference) Y to the explicit type (variable or reference) X and then obey the controlled clause if the value conformed. If the value did not conform then no assignment would occur, the condition would be false and the controlled clause bypassed.

E7 (T) RTL/2 is fully typed and does not have free unions or subset types.

E8 (F) RTL/2 does not have such initialisation and finalisation procedures largely because it does not have dynamically allocated structures which could sensibly need them. However, the extensions proposed (E2 and E6) would make such procedures useful.

Syntactically they could be added as a pair of procedure declarations appended to the appropriate type defintion. Each procedure would take a single parameter identifying the item being created or destroyed. Requirement B1 needs additional procedures defining assignment and access. Alternatively just the names of these procedures could be syntactically incorporated into the type definition. Thus

```
RECORD name (namelist)WITH
  CONS p1, DES p2, ACC p3, ASS p4;
```

where p1, p2, p3, p4 are the procedures for construction, destruction, access and assignment. The appropriate procedures would then automatically be called at the appropriate moment.

This addition to RTL/2 would not be very difficult to implement. It could perhaps be handled by a preprocessor.

As RTL/2 stands its type definitions are global to a module and so considered static. Any initialisation associated with the four procedures would be naturally static and no initialisation procedure is necessary. Complete encapsulation of the type definition with its associated procedures and local (static) data can be easily realised in RTL/2 by making these items a self contained module.

If on the other hand the language were extended to have a deeper block structure then it would be natural to permit type definitions to be embedded within blocks and a more elaborate encapsulation would be required (see F1) entailing the definition of and automatic calling of an initialisation procedure.

F1 (F) RTL/2 does not meet this. Existing structures have both global allocation and access (within a module). There is currently no initialisation or local variables anyway. (See E8).

To meet this requirement means a significant elaboration of the syntax of the language although it would have little impact on any existing implementation apart from the implication of automatic initialisation of the structures own variables. This is just a single automatic procedure call. (E8).

Scope of access could be switched on and off by additional keywords

```
ACCESS ON name (,name)...
ACCESS OFF name (,name)...
```

where name denotes the types concerned. These 'statements' would occur in a block head and apply throughout the block apart from any inner block where they are explicitly changed. The blockhead containing the type definition would have ACCESS ON unless countermanded.

F2 (P) RTL/2 partially meets this with respect to existing structures. the same record structure can have different names for its components in different modules. Similarly for databricks. (A data brick is a named collection of static data. All static variables are in a data brick and a module may contain several such bricks). Only the name of a databrick becomes external when its variables are to be made available to a separately compiled module and not the names of the variables themselves. The variables themselves can be given different names. Name clashes between subsystems are therefore rare.

The modular structure of RTL/2 enables the encapsulation within a single module of all the management routines associated with a conceptual structure and allows access to be gained only as is required. This however only applies at the module level and to conceptual structures and not to formally defined types.

The extensions E8 and F1 almost fulfill the requirements for the new structures proposed. New names could be given to separately defined program components in an ACCESS ON command by optionally allowing the type name to be followed by the new names of its components in brackets e.g.

```
ACCESS ON P(A,B)
```

would mean that the components of P would now be known as A and B. This would be easy to implement.

F3 (T) RTL/2 meets this now and with the extensions proposed.

F4 (P) Libraries are formally outside the scope of RTL/2 and more

concern the support system. Access to libraries is via external declarations. The standard input-output procedures are normally held in a program library. A brick external to a module looks the same whether it is in a library or in a separate module belonging to the user.

F5 (P) Libraries are formally outside the scope of RTL/2 and more concern the support system. The following points are of interest.

An RTL/2 module can contain an external description of a brick (e.g. procedure) actually in the module. This enables the creation of a common interface definition to be used (by file concatenation) by all the modules in a subsystem. This removes the management problems of multiply defined interfaces.

Module interfaces (in some implementations) are rigorously checked at linkage time. This has proved valuable.

RTL/2 libraries include routines written in other languages - these routines also include information for interface validation.

F6 (P) Libraries are formally outside the scope of RTL/2 and more concern the support system.

The data brick of RTL/2 has many attributes of the compool and can be used to control access to data shared between many separate modules.

F7 (P) Although formally outside the scope of the RTL/2 language standard machine independent interfaces are defined for stream input-output and error handling. Because of the vagaries of different operating systems standard interfaces for task management, timing, file manipulation and record input-output have not been formally defined. Nevertheless for our own operating systems written in RTL/2 such interfaces are informally defined.

G1 (P) RTL/2 almost fully meets this requirement. It provides features for:
sequential control - BLOCK - ENDBLOCK
conditional - IF THEN ELSEIF ELSE END
iterative - FOR or TO or WHILE DO REP
recursive - procedures are recursive.
Moreover these features are simple and well structured. The only blot is the old fashioned SWITCH (see G3). But the keywords can be pronounced, no dangling elses and simple iterative statements modelled on Algol W rather than Algol 68 (see G4).

Parallel processing and asynchronous interrupt handling are both described by the creation of independent tasks. Each task is made from a main procedure and stack both fully written in RTL/2. (You can actually declare a STACK in RTL/2). The actual creation of the task from these components is outside the scope of the language and is handled by the operating system. Task interaction, semaphores and events are also all handled by the operating system. Features of RTL/2 which are particularly useful for multi-tasking are the SVC databrick which provides each task with a private copy of system (or subsystem) wide data plus the inherent reentrant nature of code not directly writing to non local variables. These features have been especially valuable in writing compact, reentrant input-output libraries. (See B10).

Exception handling is a weaker feature. It is done using label and

procedure variables and so the language can be said to fail in this respect since there are no special control structures but just data types. New control structures (akin to ON of PL/1) would be needed to meet this requirement (see C7).

G2 (P) RTL/2 provides too much here. It has a GOTO as required but also includes switches (instead of case statements) and general label variables.

The replacement of switches by a case statement is considered under G3 and the removal of label variables and the addition of exception handling control structures is considered under G1.

It would also be necessary to restrict GOTOs to the most local scope. This would be a trivial alteration and the language would then fully meet this requirement.

These changes might have a significant impact on existing programs.

G3 (P) RTL/2 has a control structure for selection of alternatives on the value of a condition but not for subtypes from discriminated unions (no such types). It includes an unstructured switch statement which branches on a computed choice to labelled alternatives but his is not really adequate.

The simple conditional is basically

```
IF condition THEN ... END
```

or

```
IF condition THEN ... ELSE ... END
```

with a concatenated form ELSEIF to avoid most multiple ENDs. The END is the structured match to the IF and there are no dangling ELSEs. (This does not meet the TINMAN requirement since an ELSE need not follow each THEN - we could make it (trivial) but I reject this requirement). It should be pointed out that RTL/2 does not have Boolean types per se but the extensions proposed elsewhere will cover this (A2, B6).

The requirement for the selection of subtypes from discriminated unions is met by proposals in E6.

The following form of case statement (to replace switches) has been proposed:

```
SELECT expn OF
     CASE value (,value) ...:
          action
     CASE
     .
     .
     .
     ELSE
          escape action
END
```

This meets the requirement and would be easy to implement. There would be little effect on existing implementations apart from the mechanical effort in removing horrid old switches from programs.

I am not convinced that Zahn's device is 'state-of-the-art' just yet.

G4 (P) RTL/2 generally satisfied this. It provides three forms of iteration

```
FOR i:=a BY b TO c DO ... REP;
```

```
    TO n DO ... REP;
    WHILE b DO ... REP
```
In the first the explicit control variable is local. In the first two cases entry is only allowed at the head of the loop. The second two cases provide common special cases.

Requirements not satisfied are that the termination condition should appear anywhere and that the WHILE form should only be entered at the block head. The latter failing is easily remedied by a trivial compiler modification.

The requirement for the termination condition to appear anywhere is less easy. I know of no language with a nice solution to this. Perhaps the neatest solution is to define an infinite loop
```
    LOOP ... ENDLOOP
```
or more in RTL/2 style
```
    FOREVER DO ... REP
```
plus an EXIT command. Termination at any point can then be provided by for example
```
    FOREVER DO
        .
        .
        .
        IF condition THEN EXIT END;
        .
        .
    REP;
```
although the group THEN EXIT END is rather laborious.
It should be noted that one can write
```
    LET FOREVER = WHILE 0=0;
```
in order to partially create this effect although not very efficiently.

The extensions suggested here would be trivial to add and have no effect on existing programs.

Finally it should be added that the final value of the control variable can be passed out of the loop by assigning it to a more global variable at the end of the loop. This is, however, not very efficient since the assignment would be performed on every iteration. An extension to improve this would be to allow an optional termination sequence in the FOR kind of iteration. It is not clear where best to put this. A possibility would be a keyword FINALLY which with the REP embrace the termination sequence. This would be easy to implement.

G5 (T?) All procedures in RTL/2 have the same recursive storage mechanism and the lexical nesting of procedure declarations is forbidden. RTL/2 therefore meets this requirement.

It might be added that recursion has proved very useful on occasion and can help to preserve reentrancy.

Note that TINMAN does not explicitly state a requirement for lexical nesting of procedures at all. If it implies such a requirement then RTL/2 does not meet it. Assuming this is the case then the extension to RTL/2, whilst conceptually straightforward, would be very expensive to add to existing implementations.

G6 (P) This is strictly outside the scope of the language and concerns the support and operating systems. A brief outline of some of the features of RTL/2 which are of interest in this area are given in

G1.

The standard operating system features normally accessible to RTL/2 programs do however include a parallel processing capability. Tasks can be created and terminated by procedures MAKETASK and KILL. Tasks can gain exclusive use of resources by procedures SECURE and RELEASE. All tasks in RTL/2 are at the same level and there is no hierarchy of tasks. The explicit stack feature of RTL/2 ensures that run time storage is reasonably under the control of the programmer. It is not possible for the same stack to be simultaneously used by more than one task and so the problems of multiple task creation in recursive and reentrant procedures do not arise.

Thus the operating systems seem to meet the requirements even though the language itself does not.

G7 (P) RTL/2 almost fully meets this through the use of label variables and procedure variables. (Label variables were added to RTL/2 solely for this use).

Users can distinguish recoverable errors and unrecoverable errors. In the former case a user specified procedure is called and in the latter control passes to a user specified label. These procedure and label values are kept in procedure and label variables in an SVC data brick and are therefore independently defined for each task in the system.

There should be no halts beyond the users control. All exceptions such as stack overflow, array subscript violation, arithmetic overflow can be handled by the user. The user can also signal his own exceptions.

The primitive nature of the facilities provided enables the user to program the cases mentioned. He can get out of an arbitary nest of control and intercept it at any level by specifying a label value at the required level. Different actions within different scopes can be obtained by specifically preserving and restoring the label and procedure values in local variables. Transfers of control can be forward (or backward) or out of a procedure directly. Transfer of data to the exception handling mechanism is primitive - a single integer value giving the reason for failure.

The system outlined above fails to meet the requirements fully in the following areas. It probably does not allow enough data to be transmitted, it does allow exit from a procedure other than via the parameter mechanism and, of course, it uses unstructured label variables which are open to abuse. In particular, however, it seems to work reasonably well and does not get abused - perhaps because its excessive use is rather hard work.

To fully meet the requirements in view of G2 (no label variables) means completely redesigning the system. See also G1 and C7.

G8 (P) This is strictly outside the scope of the language and concerns the support and operating systems. The normal RTL/2 systems provide facilities which meet most of these requirements.

Procedures DELAY, ALRMCLK etc. allow delay on any path until a specified interval has elapsed or time been reached. Procedure NEWPRIO permits priorities to be changed. Other synchronization can be performed through exclusive access procedures SECURE and RELEASE (G6) or event signalling procedures SET, RESET and WAIT. Hardware interrupts appear

to normal application programs as the occurrance of events. There is no implicit evaluation of program determined situations.

The requirements are not met in the following areas. Hardware interrupts appear as events and not as exceptions. There is no simulated time only real time. I am sure that the addition of simulated time (if I knew what it meant) would be straightforward. To make hardware interrupts appear as events would be less easy. They could be made to appear as unrecoverable errors although this would be useless since one could not then return to the interrupted activity. Anything else would be difficult and could well interfere with execution efficiency.

H1 (T) RTL/2 fully meets this. It is a free format language, uses a semicolon as explicit statement identifier, allows mnemonic identifiers, is conventional and easily parsed and does not permit abbreviations. There are no ad-hoc special cases and it is syntactically unambiguous.

H2 (T) RTL/2 fully meets this. Operator hierarchies, precedence rules and keyword forms are all fixed and cannot be extended or altered.

H3 (T) RTL/2 meets this. The reference, publication and program writing character set are identical and comprise 59 characters of the IS07 set. It does not use 'national' characters in a potentially ambiguous situation. (National and other e.g. $ and & (or & and $ according to which side of the Great Pond you sit) are considered identical).

Translation from publication set to restricted set is not necessary since they are identical.

H4 (P) RTL/2 meets this apart from the requirement for a break character in identifiers and literals. It does, however, use precisely the suggested technique for long strings although this has caused some difficulties in machine code inserts accessing strings.

The addition of a break character in identifiers and literals would be trivial. A new character such as underline would be used rather than a space.

H5 (T) RTL/2 fully meets this. The end-of-line terminates all lexical units.

The end-of-line and similar characters can be included in strings by putting their numeric value in a # sequence. By naming this numeric value, program clarity is preserved.
Thus

```
LET NL=10;
```

with

```
"A#NL#B"
```

denotes a string consisting of the characters 'A', newline, 'B'.

A whole sequence of values can be placed between the # characters in free format. This leads to an extension which gives an alternative notation for long strings

```
"THIS IS A LONG#
#LITERAL STRING"
```

The # sequence, in this case, merely contains the layout character 'newline'.

RTL/2 meets this. All keywords are reserved and cannot be used as identifiers. They are informative but not too long. They do not get confused with identifiers. There are 57 of them which is not too many and does not inconvenience the programmer in his choice of identifiers. The various extensions proposed will increase this number but it should remain acceptable. Comments are introduced by the single character %.

I presume that the fact that user defined record class identifiers are used in the same syntactic manner as the inbuilt modes REAL etc. does not conflict with this requirement. If it does then the requirement conflicts with E2 anyway.

H7 (P) RTL/2 very nearly meets this. Comments are enclosed between % characters and must be all on one line. They cannot appear inside other lexical units (other than in a # sequence in strings) but otherwise can be freely placed.

The requirement is not fully met because a closing % is always required - this can be irksome. Of course comments cannot contain the character %.

The requirement would be more nearly met (as suggested in TINMAN) by making the closing % optional. This could be easily done (and would mean merely suppressing an error message in the existing compiler).

H8 (T) RTL/2 fully meets this. Each opening parenthesis is matched by a unique closing one. Thus

| opens | closes |
|---|---|
| " | " |
| % | % |
| PROC | ENDPROC |
| DATA | ENDDATA |
| IF | END |
| DO | REP |
| BLOCK | ENDBLOCK |

H9 (T) I think RTL/2 meets this. Certainly both function calls and array access use round brackets so that table look up can be interchanged with function evaluation without any change to the calling program.

An interesting extension might be to add an alternative notation for calling a function with only one parameter. This is

parameter. functionname

so that access to record conponents could similarly be interchanged with function calls. POP-2 uses this notation. It would be easy to add to RTL/2.

H10 (T) RTL/2 fully meets this. It distinguishes := from =. Although := is used for initialisation in data bricks this is a widely different context from procedure bricks and there is no confusion. The other cases of dual meanings are similarly widely separated in context. These are

| | |
|---|---|
| # | not equals, string inserts |
| . | decimal point, record selector |

I1 (P) Some areas of RTL/2 are not fully portable. They are

1. Range of integers and precision of fractions.
2. Range and precision of reals.
3. Order of evaluation od dyadic operands.

4. Length of stack.
Extensions already proposed (A3, C1, D4) would clear the first three. I cannot conceive of a solution for number 4.

The general philosophy of RTL/2 was that there should be no defaults.

I2 (P) RTL/2 partly meets this through the OPTION statement. This enables the user to choose for example fast or slower and more compact code. The default provides what seems best for that implementation and the option does not affect the logic of the program.

The examples given have not been implemented. To do so would greatly affect the existing implementations.

I3 (P) RTL/2 only very slightly meets this. The option statement allows the user to specify features of the object machine (e.g. has floating point hardware or not) either from within or outside the program. However, this is a separate feature and no compile time variables are involved and the program cannot interrogate them at compile time.

This requirement is related to I4 and its fuller satisfaction is discussed there.

I4 (F) RTL/2 does not meet this. This would best be satisfied by a macro preprocessor with macro time variables determining the configuration and which as well as selecting the appropriate RTL/2 text also generated appropriate OPTION statements so that the compiler is informed of the object machine configuration.

Such a solution would have no impact on existing implementations and would be fairly easy to implement.

I5 (-) I do not understand the requirement.

I6 (P) This is outside the definition of RTL/2 but nevertheless the modular construction of the compilers leads to this requirement almost being fulfilled.

All our compilers have the same structure:

- driver
- front end
- interface
- back end

The front end is common to all compilers. The driver interfaces with the operating system and controls overlays. The back end depends on target machine. The interface contains the symbol tables which control almost all the constraints mentioned in this requirement. There have been several versions of this interface with different table sizes for different configurations but portability considerations are forcing us to use just one version for all situations on all implementations. The compromises are difficult but it can be done and the greedy users complain at times.

Typical values are

| | | |
|---|---|---|
| array dimensions | : | 8 |
| length of identifiers | : | 31 |
| number of identifiers | : | 600 |

To fully satisfy this requirement mostly means writing these values in the language definition.

However, there is one minor point. The compiler analyses expressions recursively and the constraint on bracketing depends on the stack size available. A small addition would need to be made to monitor this recursion explicitly and fail at a fixed level.

I7 (P) RTL/2 compilers do not have language restrictions dependent on the object environment. Nor do they report when its resources are exceeded although they do list resources used. The compiler cannot tell if the stack will overflow before run time in the general case. This requirement cannot be met by any obvious extension to RTL/2 or its implementing software.

J1 (P) This requirement was certainly a key objective in the design of RTL/2 and has been well met although clearly perfection is difficult. Basic language support routines (for procedure entry, array subscript checks etc.) are only one or two hundred words and being reentrant are shared among all tasks. The language was designed so that no support was required for features such as dynamic array allocation. I/Ø support is written in RTL/2 and part of the library and only included in the system if needed.

Process scheduling, overlay control, file management etc. are outside the scope of the language definition itself.

All compilers include a (table driven) optimisation phase which caters for commonly occuring special cases. However, a point of diminishing returns is soon reached.

There are few places where unused language features impose costs on otherwise simple situations. The only ones I can think of are
i) the full generality of label variables is never used and the compiler is burdened by these features and does not treat all the common cases of label value manipulation as well as it might,
ii) rounding rather than truncation on fractions makes the code less efficient than would otherwise be the case and I don't believe anybody needs the rounding anyway. Both of these blemishes would be removed by proposals already made (G2, B5).

Extensions to RTL/2 such as dynamic arrays (A6) would need care to ensure that this requirement remains well met.

J2 (T) As far as is known RTL/2 compilers meet this requirement. There are, however, cases where illegal programs fail in slightly different ways because of optimisations.

J3 (P) RTL/2 allows machine language insertions which meet most of the requirements. Machine language insertions are encapsulated within CODE ... RTL brackets and are only allowed in the system (or full) version of the language. RTL/2 variables and identifiers can be accessed in terms of their RTL/2 name by the use of a distinctive prefixed trip character. Access is constrained to ensure that scope rules are not broken and redundant information is required to ensure that the user distinguishes between for example local variables and those in different data bricks.

Machine language insertions are used for writing device drivers, context switching in the schedular, some interrupt handling and machine

dependent mappings outside the scope of the language. The distinction between an application language (in which access to the machine language is forbidden) and a system language in which it is allowed helps to prevent the unnecessary use of machine language.

The requirement that machine language insertions only be allowed in the body of compile time conditionals dependent on the object machine is not met since RTL/2 does not have these features. The proposal I4 for a preprocessor to provide compile time facilities would enable this to be met.

J4 (F) RTL/2 does not meet this requirement at all within the language although, of course, it is possible to write modules containing structures so that all access is via procedures and the underlying structures themselves are not visible.

The addition of facilities to RTL/2 itself in order to satisfy this requirement, whilst conceptually obvious, would be tedious and add considerably to the complexity of existing compilers. A preprocessor might add some features more easily.

J5 (F) RTL/2 does not meet this. An extension to do so would not be difficult within a single module. To do so between modules involves interaction with the library mechanism and is outside the scope of the language itself.

The option statement offers one way in which the compiler could be told to treat certain procedures as open. Although such an extension would not be difficult it would have a significant effect on existing implementations.

K1 (T) RTL/2 meets this. The RTL/2 language features do not require an operating system. However, if some of the features such as tasking (F7, G6, G8) were fully integrated into the language the meeting of this requirement would become extremely difficult. The compiler would need to be augmented by what is, in effect, an automatic system generator.

K2 (P) An RTL/2 program is comprised of separately written modules. Currently the module is a unit of compilation. However, the statement to this effect could be deleted from the language specification and the choice left to the implementor.

RTL/2 enables interfaces to be will defined via EXT specifications describing bricks outside the module being compiled and ENT tags making bricks in the module accessible outside. EXT specifications contain the information necessary for full type checking.

K3 (P) Each RTL/2 implementation has support packages and debugging systems. These vary between systems largely because of the cost of their development. They are written in RTL/2 with occassional code statements for such purposses as mapping parts of the stack. The best system is that for the IBM 370. In this a separate task decodes the task to be debugged and can provide a forward and retro trace, parameter value trace, count of procedure calls etc. This task uses a separate stack to avoid interference with the stack being debugged. The production of such a package across all implementations is largely a question of cost and organisation of suitable hooks into the object code which do not degrade the performance on the object machine. Such hooks can be controlled via the option statement.

RTL/2 implementations sometimes use the linkage editor of a pre existing operating system. In such cases a separate 'linkage verifier' is available which enables the interfaces between the separate modules to be checked for consistency. In cases where the operating system is written in RTL/2, this verification is a built-in feature of the linkage editor. Data input to the linkage editor is revolting and would need much revision to meet the requirement for a 'consistent easily used user interface'. This is, however, likely to be done as part of a revision of RTL/2 system generation aids.

K4 (P) Since this requirement is not precisely stated it can hardly be fully met. Program post-mortem analysis and diagnostics is discussed in K3. Program reformating is not provided (we were going to but could not agree on a specification - it needs reconsideration). Cross-reference generation is always available as a compiler option. More could be provided - particularly in the compiler front end since its benefits would then be felt by all implementations equally. Reformating and more listing control could be easily added.

K5 (F) RTL/2 completely fails in this area - as do most languages. I do not feel able to propose much here other than that there be a special form of comment. In the immediate future they would be treated as normal comments.

L1 (T) RTL/2 has remained frozen since its final release in 1973. All compilers written since then have incorporated a common front-end and no supersets have been allowed. We have retained strict control of the source of the front end and have not made it available to implementors. We bootstrap for them.

L2 (T) As for L1. All implementations have been required to implement the full language. Some minor difficulties loom ahead regarding the use of manufacturers linkage editors which differ in their ability to satisfy cross references.

L3 (P) RTL/2 compilers are not very fast but neither are they too slow for practical use. They might be too slow for 'student jobs'. They are all of similar size and comfortably run on a 32K machine with some backing store. Since the language is strict many programs fail to compile and only the front end of the compiler is invoked. Such uses of the compiler are fairly fast. When a program is correct the back end, including optimisation, is invoked and this takes somewhat longer. Trivial programs take a trivial time to compile (only compiler overlay time).

Users have little control over the trade off between compile time and runtime - some compilers allow the invocation of part of the optimisation to be optional. This could be extended to all compilers at some cost.

L4 (P) The structure of RTL/2 compilers is outlined in I6. The machine independent parts are reasonably independent of the generators. The front end is completely machine independent and is absolutely unique. There are back ends in existence for about ten different target machines. More could (and will) be written to more fully meet this requirement.

L5 (T) RTL/2 compilers meet this. They are written in RTL/2 and self hosting is often but not always performed. Within ICI almost all compiling is done on machines other than the final target machine. The

compiling machine is often a larger machine of the same type but compilation on IBM 370 for PDP-11 is common.

I6 (T) As far as we know RTL/2 compilers meet this. They do not attempt to automatically correct errors detected at compile time.

L7 (P) RTL/2 compilers produce their diagnostic messages via a standard interface (see I6). As a consequence the messages are uniform across implementations. They include an English language message and references to program identifiers. (Some compilers configured for small machines produced only a cryptic message - these are being phased out and replaced by those producing the standard messages). Messages produced by translator confusion are few. The unique closing keywords help here (see H8).

RTL/2 compilers do not meet the requirement for the production of code generating run time exception conditions corresponding to source program errors. If a program is in error no code is generated. The meeting of this requirement (by block or procedure) would be reasonably straightforward.

Warning messages are produced when certain unlikely constructions are used but which are nevertheless legal.

Since the diagnostic messages are uniform across implementations it would be a trivial matter to add these to the language defintion in order to meet the stated requirement.

L8 (T) The RTL/2 language as it now is plus extensions proposed here implies no particular style of translator.

L9 (T) This is met. All RTL/2 compilers are entirely written in RTL/2 (application version as opposed to system version and so there is no embedded machine code). The overall structure is outline in I6. The front end produces output consisting of i) symbol tables ii) coded form of the program as a vector. This is relatively easily translated by the individual back ends into the required object code. The cost of developing a reasonably straightforward back end is one man year of which a significant portion is learning i) RTL/2, ii) the target machine iii) the compiler interfaces. Bootstrapping on to object machines is easy; the minimum configuration which seems sensible is 32K plus some backing store. Smaller machines have been used.

Of the extensions proposed in this document many (including all those which might be implemented by a preprocessor) would add nothing to the cost of developing new backends. Other extensions, particularly those concerning storage allocation, block structure, exception handling could well double or treble the cost of new back ends.

M1 (P) RTL/2 was composed from the state-of-the-art in about 1969. That meant Algol 68, avoiding its worst excesses and academicism, adding special features such as fractions and a modular structure. Considerations of cost of implementation meant that the language had to be slim in order for the project to show a return on a modest capital investment ($1M) in a reasonable time. Slimness ensured that the design was coherent.

Inevitably the TINMAN requirements reflect 1976 state-of-the-art. For RTL/2 to be extended to meet these as outlined here would need care. The overall slimness would be lost and there would be a danger of losing

coherence. However, the lexical framework of RTL/2 meets the requirements of section H very well indeed and so RTL/2 provides a good starting point for meeting TINMAN from a low level viewpoint.

M2 (P) The semantics of RTL/2 are defined informally. There are few ambiguities and it is hoped to revise the coumentation to clear those that do exist in due course. The use of a single common front end to all compilers means that the same interpretation is, however, applied to all implementations. No attempt has been made to produce a formal specification.

M3 (P) To pretend that any set of language documents is perfect would be foolish. In the case of RTL/2 the documentation is excellent and includes
i) RTL/2 language specification. This is a semiformal description for reference purposes. It includes an Algol 60 like description with syntax in a variant of BNF and semantics given in English, examples of each construct, list of keywords, character set etc.
ii) RTL/2 training manual. This is a complete tutorial document with worked examples and problems (with answers) for the student.
iii) RTL/2 system standards. This describes input-output and error recovery and other standards which whilst not part of the language should be uniformly available on all implementations. It is intended primarily for the writer of support systems.
iv) RTL/2 standard stream I/0. This describes the input-output standards from the users point of view.
v) RTL/2 Hints. This is an ad-hoc document containing miscellaneous notes on how to get the 'best' out of RTL/2.
vi) RTL/2 Language Reference Card. This is a folded pocket card containing a brief description of the language. It includes examples, tables of keywords, operators, etc.

If RTL/2 were extended then these documents would be updated to the same standard.

M4 (-) RTL/2 is currently controlled by Imperial Chemical Industries Limited. All implementations have to pass tests supplied by ICI before release is approved. The language logo is trademarked.

The RTL/2 language is being considered by the British Standards Institute (BSI) as a standard language for 'Industrual Real Time'.

M5 (-) RTL/2 software is supplied and maintained by agents or licencees appointed by ICI.

M6 (-) As for M5.

## III. LANGUAGE FEATURES NOT NEEDED

There is very little in RTL/2 that has not been used to meet some or other of the requirements in Section II. The following are on the borderline.

1 Fractions and double length forms.

The elaborate double length fixed point forms of RTL/2 do not seem to be explicitly required. However, requirement A4 demands fixed point arithmetic and if the existing RTL/2 fractions were felt to meet this satisfactorily then the double length forms ought to remain.

2 Shift operators - logical.

These have not been explicitly required. However, if requirement J4 were to be met using a preprocessor then shift operations would be needed at the intermediate level. They should be made invisible at the language level.

3 Shift operators - arithmetic.

These are used for fixed point scaling. If one were to revert to RTL/1 type fixed point with automatic scaling then these operators could be dispensed with. However, using RTL/2 fractions as they are they need to be retained.

4 Label variables.

These should be deleted and replaced by new control structures.

5 Stack variables.

Even if stack bricks are retained, stack variables seem of little value.

6 Titles.

These usefully label a complete module for documentation and record purposes and should be retained.

## IV DISCUSSION AND SUMMARY

The comparative evaluations are summarized in the table below.

| SECTION | TOTAL | T | P | F | - |
|---|---|---|---|---|---|
| A DATA AND TYPES | 7 | 1 | 4 | 2 | |
| B OPERATIONS | 11 | 6 | 4 | 1 | |
| C EXPRESSIONS AND PARAMETERS | 9 | 2 | 4 | 3 | |
| D VARIABLES, LITERALS AND CONSTANTS | 6 | 1 | 4 | 1 | |
| E DEFINITION FACILITIES | 8 | 3 | 2 | 3 | |
| F SCOPES AND LIBRARIES | 7 | 1 | 5 | 1 | |
| G CONTROL STRUCTURES | 8 | 1 | 7 | 0 | |
| H SYNTAX AND COMMENTS | 10 | 8 | 2 | 0 | |
| I DEFAULTS, CONDITIONAL COMPILATION etc | 7 | 0 | 5 | 1 | 1 |
| J EFFICIENT OBJECT REPRESENTATIONS & MACHINED DEPENDENCIES | 5 | 1 | 2 | 2 | |
| K PROGRAM ENVIRONMENT | 5 | 1 | 3 | 1 | |

| | | | | | |
|---|---|---|---|---|---|
| L | TRANSLATORS | 9 | 6 | 3 | Ø | |
| M | LANGUAGE DEFINITION, STANDARDS AND CONTROL | 6 | Ø | 3 | Ø | 3 |

Note that category U has not been used largely because the author is totally familiar with RTL/2 and has access to all documentation. In a few cases a don't know or irrelevant category (-) has been used. This covers one case (I5) where the requirement is confusing and some of the M section where the requirements are obviously not expected to be fulfilled at this time anyway.

Before analysing the evaluations it is worth outlining reasons for the general differences and simularities one might expect between RTL/2 and a language meeting TINMAN. (This repeats M1 to some extent).

1. The overal goals of RTL/2 were similar to those for TINMAN. These included such things as portability, readability, good documentation, support, clear compiler construction etc.

2. RTL/2 was composed from the state-of-the-art in about 1969. It was after Algol 68 but before PASCAL. TINMAN reflects 1976 state-of-the-art and has the benefits of the experience of PASCAL.

3. The capital available for the development of RTL/2 was an order of magnitude less than that said to be available for TINMAN. RTL/2 is therefore much slimmer than the language envisaged by TINMAN. In its design it was not usually a question of 'would a feature be useful' but rather is it absolutely necessary'. Certain things left out of RTL/2 are clearly worthwhile given a broader capital base from which to start.

As a consequence of this background, RTL/2 would, with a few minor exceptions, need extending rather than changing in order to meet the TINMAN requirements. Many extensions would be trivial and independent of each other. A few key areas need major attention and these are discussed in detail below.

The areas of greatest match between RTL/2 and TINMAN are sections H and L with B a close third. The match with section H: syntax and comment conventions, is almost perfect and at this level RTL/2 provides a good springboard from which a language to meet all the requirements can be designed. The close match with section L: translators, reflects that the general goals were similar.

The areas of worst match are sections C and E with A and J not quite so bad. These mismatches are mostly due to the smallness of RTL/2; some also reflect the influence of PASCAL.

In the proposals for extending RTL/2 to meet the requirements, the use of a preprocessor has been suggested. This is, in some ways, an internal matter and shows how some of the extensions might be incorporated into our present compilers. If an extension can be done by a preprocessor then clearly it cannot effect the code generators and this makes it more acceptable. In practise, however, for anything other than a pilot experiment, a fully integrated compiler is necessary.

The key areas needing attention are as follows.

1 Data types. (B11, E6)

These are extensions to provide enumeration types, power sets etc. and to generally cloak a lot of integer manipulation in a more modern guise. This is the 'pre processor' area mentioned above. Not difficult but requires thought.

2 Parameter mechanism. (C7, C8, C9)

One is unhappy with these TINMAN requirements anyway. The requirement for generic procedures and multiple parameters are not convincing. This would be a lot of work.

3 Exception handling. (C7, G1, G2, G7)

This is a reasonably self contained area and entails cutting out label variables and replacing them by a structured control mechanism. Although a detailed proposal is not made nevertheless it is clear that it could be done without difficulty although existing compilers would need a lot of change.

4 Aggregate data structures. (A6, A7, D5, D6, E2, E6, E8)

This and the next area would require the greatest work. The TINMAN requirements seem to conflict regarding dynamic storage. A6 is confusing in itself and is in conflict with finding 2 on pl2 of paper P1191.

If true dynamic arrays are required then RTL/2 would need a lot of extending. If, however, arrays only need to be allocated from a domain or pool using slicing or trimming then this fits better into the RTL/2 philosophy. Extensions to RTL/2 to meet this have been considered in the past.

The addition of unions, variant records, Simula-classes would all be feasible but would mean massive changes to existing compilers.

5 Procedure structure. (G5, J5)

Allowing lexical nesting is conceptionally trivial. But RTL/2 was designed around a flat structure in the belief (borne out in practise) that you can 'get away with it'.

Extensions to meet this would change the face of RTL/2 and make it into a significantly different language. The compilers would need considerable alteration.

If the key areas described above were attended to plus the many points described as trivial in the text then one would have a language which was a cross between PASCAL and Algol 68 in RTL/2 lexical style and which met almost all of the TINMAN requirements. It is clear, however, that although some extensions could be done by adding to the existing RTL/2 compilers, nevertheless all the extensions add up to a much larger language than RTL/2 for which a compiler would most sensibly be written from scratch.

---

TEXAS INSTRUMENTS INCORPORATED
304 Wynn Drive
Huntsville, Alabama 35806

August 2, 1976

# SUITABILITY OF THE PROCESS DESIGN LANGUAGE AS AN INTERIM DoD HOL

## Introduction

This report compares the language requirements defined by the DoD HOL "Tinman" with the Process Design Language (PDL2) developed by Texas Instruments for the Ballistic Missile Defense Advanced Technology Center. The purpose of this comparison is to identify the similarities and differences between the HOL and PDL2 and to assess the suitability of PDL2 as an interim HOL.

## PDL2 Overview

PDL2 is a PASCAL based language incorporating extensions in several areas to meet the needs of BMD real time software design, implementation and testing. A compiler has been implemented for PDL2 on the Advanced Scientific Computer (ASC) which produces code for either the ASC Central Processor or the ASC Peripheral Processor.

The major extensions in PDL2 include:

- Tasking and Synchronization Primitives - Processes are likely to be implemented on multiprocessors to obtain necessary computing capacity. Even in the case of a single processor, the need for interrupting tasks by tasks of higher priority will cause a parallelism which must be taken into account in preserving data sanity. The approach chosen for PDL2 is to provide primitives suitable for the implementation of specific schemes of

tasking and synchronization.

- Variable Length Arrays - A recognized deficiency of PASCAL is the absence of variable length arrays. PDL2 permits the declaration of arrays whose upper dimension is bound when the array is allocated. PDL2 also allows variable length arrays to be passed as procedure parameters. In PASCAL the user is limited to array declarations of the form: A: ARRAY [0..10] OF REAL. In PDL2 the upper bound may be any expression such as: A: ARRAY [0..N*M] OF REAL.
- Vector Operations - Operations on vectors were added primarily to exploit the vector capabilities and parallelism of advanced computer architectures, but may also increase the code efficiency on serial computers. Vector operations also contribute to code readability. For example in PASCAL a vector operation would be written as:

```
FOR I := 0 To N DO
FOR J := 0 To M DO
FOR K := 0 To L DO A [I, J, K] := B [I, J, K] = C [I, J, K];
```

  In PDL2, this would simply be A := B + C.
- Assertion Statements - These statements provides optional run-time verification (not proof) of the programmer's proof assertion. Thus, if a program contains:

```
ASSERT I + J < 100
```

  Run-time code is generated to evaluate the expression I + J < 100 and to output an appropriate error message if the expression is false.
- Parameterized String Substitutions (MACROS) - Macros permit the

definition of in-line procedures and functions. Macros allow the user to make the decisions on trading space for speed. Macros intentionally are syntactically similar to procedure calls so that conversion from one to the other is facilitated. For example, in the compiler, there was a procedure which returned the next character in the input string. This procedure was very short and called frequently but from only a few procedures. Conversion of this procedure to a macro resulted in a significant speedup in the compiler.

- An Escape Statement - A frequent criticism of structured programming languages is the inefficiency involved in terminating a loop which contains rather complex logic. The escape statement provides an efficient disciplined method of loop termination. The approach of PDL2 for an escape is to allow any statement to be labelled. Occurrence of the statement ESCAPE LABEL within the labelled statement causes a transfer of control to the next statement after the statement labelled LABEL.
- Commons - These were added to PDL2 for two main reasons. First of all, they were necessary to communicate with Fortran and secondly, they provide an own variable capability in PDL2. The implementation in PDL2 is such that most of the usual problems with commons are avoided. The common is declared only once at a level such that it is a root of all subtrees which reference the common. Individual procedures which reference the common must explicitly request access to the common with the ACCESS statement.
- PP Code Generation - The compiler also generates code for the ASC peripheral processor to facilitate operating system implementation.

Machine specific features are also defined in a general manner for portability. For example, the CR file is definitely a machine dependent feature, yet it is simply defined as a common. The compiler must know about the CR file and treat it specially, but compilers are always machine specific.

HOL Requirements

The requirements for the DoD HOL are presented and addressed in this section. First, the HOL requirements as stated in the "Tinman" document is given followed by a discussion of the PDL2 similarity or difference. A summary is given at the end of each major requirement area.

A. Data and Types

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable, and component of composite data structures be explicitly specified in the source programs.

A2. The language will provide data types for integer, real (floating. point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

PDL2 satisfies requirements A1 and A2. PDL2 is typed with all types determinable at compile time, and all of the HOL data types are provided except fixed point real. Fixed point arithmetic is almost redundant since it provides essentially the same capability as floating point real. Admittedly, it is a convenience in some applications, and may result in improved accuracy. There is nothing contradictory in PDL2 which would prevent addition of this feature.

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

PDL2 allows only two precision for floating point arithmetic: normal and extended. It can be set global to scope by declaring REAL to be LONGREAL or vice versa, however, once redefined precision of individual variables is fixed. While the HOL treatment is preferable, the PDL approach is certainly a reasonable choice. The Tinman also does not describe how the precision will be specified. In the Tinman approach, it is also desirable to permit specification of precision for specific types. If this were allowed, in actual usage, HOL and PDL2 would probably be equivalent, since a programmer would probably define a standard precision, an extended precision and choose one of these for each variable.

A4. Fixed point number will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.

Given that fixed point numbers are allowed, this is certainly the correct interpretation.

A5. Character sets will be treated as any other enumeration type.

This is a desirable feature for transportability. PDL2 does not provide this capability. However, for experimental purposes, it should not prove to be a severe handicap.

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

PDL2 with minor exceptions satisfies the requirements. The first exception is that the dimension is a part of the type in PDL2. In PDL2, the indefinite dimension was introduced to avoid problems with generic routine and appears satisfactory. What's more, HOL certainly will not allow assignment of a ten word array to a five word array. Whether this is called a type violation or something else is unimportant. The second exception is that in PDL2 the lower

bound is not only fixed, for variable length arrays, it is zero. This has proven to be a good restriction for implementation purposes, and it also permits generation of a better code. The restriction was adopted for several reasons. In the first place, PDL2 permits array cross sections. What is the lower bound of a cross section? Cross sections are valuable, and having zero as the lower bound of variable length arrays and cross section has proven to be useful restriction. Secondly, most programmers use either zero or one as the lower bounds for almost all arrays. When procedures written by different programmers are integrated this is usually a problem. This problem is eliminated by fixing the lower bound. Since the zero lower bound permits generation of better code and at the cost of wasted storage location includes the one lower bound case, zero was chosen as the lower bound.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

PDL2 fully satisfies this requirement.

In summary with respect to data and types PDL2 and HOL are similar in that both:

- are strongly typed languages.
- have similar record structure.
- have variable length arrays allocated at scope entry.
- have almost the same built-in types.

PDL2 and HOL differ in that:

- PDL2 does not support fixed point arithmetic.
- PDL2 has only two floating point real precisions.
- PDL2 does not treat character sets as any other enumeration type.

B. Operations

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a varaible, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

PDL2 essentially meets this requirement. The only exception is that PDL2 does not have a union type.

B2. The source language will have built-in operation which can be used to compare any two data objects (regardless of type) for identity.

PDL2 does not fully satisfy this requirement. Comparison of records or arrays is not allowed in general. Comparison of strings which are packed arrays of characters is allowed. By means of the type transfer, PDL2 has an equivalent capability which is slightly more troublesome to use.

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration.

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

PDL2 satisfies both of the above requirements except as previously stated, PDL2 has no fixed point data, and exponentiation is provided by a built-in function rather than an infix operator. This was adopted because there does not appear to be a agreement whether exponentiation should be evaluated left to right or right to left. As a function the question does not arise.

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

B6. The built-in Boolean operations will include "and", "or", "not", and "nor". The operations "and" and "or" on scalars will be evaluated in short circuit mode.

PDL2 fully satisfies the above criteria with the exception that the "nor" operation is not built-in and "and" and "or" are evaluated in short circuit mode for conditionals but not in Boolean assignments.

B7. The source language will permit scalar operations and assignments on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

PDL2 permits data transfers between arrays of identical structure but not between arrays or records with identical logical structure.

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversions operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

PDL2 does not fully satisfy the requirements. There is automatic conversion from INTEGER to REAL, REAL to LONGREAL, and LONGREAL to REAL. In addition, PDL2 does not have union types. An explicit conversion is provided to go from REAL to INTEGER (TRUNC) and CHR and ORD are provided to convert to and from character representation.

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

PDL fully satisfies this requirement.

B10. The base language will provide operations allowing programs to interact with files, channels, or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

PDL2 satisfies little of this requirement. Capability is provided to read and write both sequential and direct secondary files. However, the interactions with files or channels are all handled by the compiler or by the operating system.

The PP compiler does provide for interaction with the CR file and absolute memory. This capability is definitely installation dependent.

B11. The language will provide operations on data types defined as power sets of enumeration types (See E6). These operations will include union, intersection, difference, complement, and an element predicate.

PDL2 exceeds this capability in that power sets of subranges of integers are also allowed. Restricting sets to enumeration types is good in that it facilitates typing of constant sets.

C. Expressions and Parameters

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

PDL2 fully satisfies this requirement.

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

This requirement is sufficiently vague that it is difficult to determine whether PDL2 satisfies the requirement or not. Certainly PASCAL has too few levels of hierarchy in that statements which appear perfectly reasonable are actually errors. PDL2 has as few levels as is consistent with readability and accepted usage.

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.

PDL2 satisfies both of the above requirements.

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handing, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring applicable only to parameters).

PDL2 satisfies this requirement with the exception that array references are not the same form as other parameter references.

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

In PDL2 formal and actual parameters always agree in type and the number of dimensions for array parameters are determinable at compile time.

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

PDL2 violates several parts of this requirement. The first two types of formal parameters correspond to PDL2 value and reference parameters. In PDL2, however, value parameters can be utilized as temporaries. The restriction on assignment could be useful in a single process system allowing the actual data item to be used instead of making a copy. This would be an extremely unsafe implementation in a multiprocessing system. PDL2 has no parameters in the last two classes. Passing procedures as parameters was felt to be unsafe in a multiprocessing system.

C8. Specification of the type, range, precision, dimension, scale and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.

PDL2 provides an equivalent capability through a variety of means for those specifications that are applicable. Optional type in PDL2 is achieved by means of the type WORD. A formal parameter of type WORD will match any type. An unknown dimension is signified by an upper bound of question mark. Precision is included in the type and scale and format are not applicable.

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

PDL2 currently does not provide for a variable number of arguments with the exception of built-in procedures such as READ and WRITE. This is a useful feature, however, lack of it seldom causes much difficulty.

In summary, PDL2 meets most of the requirements of HOL with respect to expressions and parameters. The major differences are:

- PDL2 uses different notation for array references and function calls.
- PDL2 does not have formal parameter classes for specifying control action or procedure parameters.
- PDL2 does not permit a variable number of arguments.

D. Variables, Literals and Constants

D1. The user will have the ability to associate constant values of any type with identifiers.

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).

PDL2 fully satisfies the above criteria.

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

PDL2 does not have capability for initialization. In a block structured language, initialization must be done each time the allocation scope is entered. For practical purposes, a procedure will actually have to be executed to perform initialization regardless. Initialization should be specified in the same types of statements as the body of code. Therefore, what is really being considered

is where initial values occur. To allow initial values as part of the declaration contribute to readability but that's about all.

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables.

PDL2 does not meet this requirement. Range of variable can be specified for all numeric variables or enumerated types, but it is never required. This feature will prove troublesome to the user, but should result in more reliable software. Again, PDL2 has no fixed point variables.

D5. The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.

PDL2 satisfies this requirement completely.

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only effect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

PDL2 does have pointers. They are not as safe in their use as any other variables are. What is more, it is not clear that prohibiting pointers to data structure where allocation scope is narrower than that of the pointer variable is enough to insure safety. What is to prevent reference to data which has been explicitly released? The implication is that the user cannot explicitly dispose of data. With this restriction, PDL2 is safe.

To summarize, with respect to variables, literals, and constants, PDL2 satisfies HOL requirements except with regard to initialization and range specification.

E. Definition Facilities

E1. The user of the language will be able to define new data types and operations within programs.

The user can specify new data types in PDL2 but not new infix operations. Procedures and functions provide the capability for defining new operations. While this is all that is necessary, infix operations do contribute to readability.

E2. The "use" of defined types will be indistinguishable from built-in types.

This is true in PDL2. In fact, this is exactly how built-in types are implemented - by simply defining them internally at compiler initialization.

E3. Each program-component will be defined in the base language, in a library, or in the program. There will be no default declarations.

PDL2 satisfies this requirement.

E4. The user will be able, within the source language, to extend existing operators to new data types.

PDL2 does not allow extending existing infix operators other than by procedure or function calls.

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

PDL2 does not provide this type of definitional facility. Clusters and monitors have been considered for possible inclusion in PDL2. Addition of this capability should not be very difficult.

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

PDL2 does allow enumeration types, Cartesian products of existing types,

and power sets of enumeration types. Discriminated union is not explicitly allowed although the capability is provided in a clumsy sort of way by variants of record structures.

E7. Type definitions by free union (i.e., union of nondisjoint types) and subsetting are not desired.

PDL2 does not allow free union or subsetting, and therefore satisfies this requirement.

E8. When defining a type, the user will be able to specify the initialization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

PDL2 does not permit initialization and finalization procedures.

In summary, with respect to definition facilities, PDL2 is close to HOL except that PDL2 does not allow defining infix operators nor does it support the idea of data with particular operations associated with them.

F. Scopes and Libraries

F1. The language will allow the user to distinguish between scope of allocation and scope of access.

PDL2 does not really distinguish between scope of allocation and scope of access in the sense intended by the HOL. On commons, however, such a capability is provided, but scope of allocation is in fact static, not that implied by the block structure of the program.

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

PDL2 does not satisfy the criteria since there is no way to limit access.

F3. The scope of identifiers will be wholly determined at compile time.

PDL2 satisfies the criteria completely.

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

PDL does support libraries and these libraries are easily accessible in the language. Since PDL2 does not have clusters, the information stored in the library is not of the proper kind to satisfy this criteria.

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

PDL2 satisfies this requirements. The PDS system does include a library system which can store procedures, functions and commons. Actually, a module can contain any text.

F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

PDL2 does not have compools. The capability for handling global data is provided by allowing individual procedures to be compiled separately and having the configuration processor stage up the appropriate declaration. This requires recompilation of the entire system when a global declaration is changed, but does permit each procedure to be separately compiled. Another advantage to this is that the compiler does not interface directly with a library and can therefore execute more efficiently. Thus, the capability suggested by compools is present, merely the implementation is different.

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

F7 was an initial goal of PDL2. Unfortunately, this goal was not fully satisfied. While machine dependent interfaces are not included in the PDL2 language definitions, they are implemented in the compiler. This may also

prove to be a difficult goal for the HOL to attain.

PDL2 is not a good approximation to HOL with respect to scopes and libraries. PLD2 does have some means for limiting access to commons but the library philosophy is somewhat different in that the compiler does not directly access the library. The necessary capabilities are present in PDL2 however.

G. Control Structure

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

PDL2 does provide structured control mechanisms for sequential, conditional, iterative and recursive control. The primitives for (pseudo) parallel processing are also provided. Exception handling is not adequately treated, however. When an exception occurs PDL2 essentially either stops with appropriate messages or indicates the exception and continues as if it had not occurred. This is recognized as a weakness of PDL2, and represents an area in which future effort will be devoted. Asynchronous interrupt handling can be accomplished in PDL2 as implemented on the ASC, however, this is machine dependent.

The problem is that PDL2 when producing code for the ASC PP is a language for writing operating systems. For such work it is desirable to work directly with machine features. To make this machine independent would have required placing unnecessary constraints on the virtual machine on which compiled programs are to be executed.

G2. The source language will provide a "GOTO" operation applicable to program labels within its most scope of definition.

PDL2 exceeds requirement G2 in that not only is the GOTO allowed within its most local scope but global GOTO is also allowed. This is not to say that use of the GOTO is encouraged. Indeed, PDL2 provides the ESCAPE as a structured

alternative to the unstructured GOTO.

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

PDL? does provide both the IF contstruct and the CASE construct. These are not constrained to be fully partitioned. In fact, while the requirement that a CASE statement be fully partitioned is reasonable, to require an "ELSE" part for each "IF THEN" is not.

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

PDL? actually provides three iterative control structures: WHILE, REPEAT, and FOR. The termination condition may appear anywhere in the loop in the form of an ESCAPE statement, and entry is allowed only at the head of a loop. Loop control variables are not local to the loop. PDL? adopted the policy that all variables be declared, and variables can be declared only for procedures or functions.

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

PDL? exceeds requirement G5 in that all procedures are potentially recursive and are treated as such by the compiler. The HOL restriction is a good one in that it attacks the limiting factor in execution speed. However, the fact that PDL? does not execute as rapidly as it could with this restriction in no way detracts from usage as an interim experimental vehicle.

G6. The source language will provide a parallel capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

PDL2 fully satisfies this requirement.

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

As previously discussed, exception handling is weak in PDL2.

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permits specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

PDL2 permits delaying of tasks only, and occurrence of the specified time or situation must be programmed explicitly. Priorities also must be explicitly programmed. The primitives were chosen such that the user is not locked into a particular priority or event management scheme. The price paid for this generality, however; is that a particular scheme must be programmed by the user. Access to the real time clock is not provided although it could easily be added as a standard function if the hardware supported a real time clock. Asynchronous hardware interrupts are not treated in any manner other than as explicitly programmed by the user.

To summarize with respect to control structure, PDL2 provides the standard control structure with differences on minor points which are really a matter of taste such as local control variables or fully partitioned conditional. The only major differences stem from limited exception handling in PDL2.

H. Syntax and Comment Conventions

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

PDL2 does not satisfy parts of this requirement. The decision was made early in the development of PDL2 that PASCAL would be maintained as a subset where possible so that development work could proceed in PASCAL while the compiler was being implemented. This led to adoption of the ALGOL drived convention that a semicolon is a statement separator rather than a delimiter. In addition the ALGOL IF-THEN-ELSE with the attendant dangling ELSE ambiguity problem was also retained. Neither of these problems is serious since a rather simple translator can be constructed to convert one syntax to the other. With the exception of the two mentioned problems, PDL2 satisfies H1.

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences.

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

PDL2 does not allow continuation of lexical units across lines, but there is no way to include object characters such as end-of-line in literal strings. This is primarily because PDL2 is implemented on a batch mode, card oriented machine, and the need has never arisen.

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

Requirement H6 was certainly an objective in the design of PDL2 and every attempt was made to satisfy that objective. The widely varying interpretation of the term "few", however, make determination of conformance to this requirement difficult to assess. With this caveat PDL2 satisfies the requirement.

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

PDL2 does allow comments, however, the end of a line does not terminate a comment. This is relatively minor again since mechanical translation from one form to the other is trivial.

H8. The language will not permit unmatched parentheses of any kind.

PDL2 satisfies this requirement.

H9. There will be a uniform referent notation.

PDL2 is an ALGOL derivative and thus uses square brackets for array references and parenthesis for functions. While it is agreed that the distinction between function calls and data references are not necessary, it is useful to be able to determine from the syntax just what something is. This is another relatively minor point, and compiler modification or automatic translation is very simple.

H10. No language defined symbols appearing in the same context will have essentially different meanings.

PDL2 satisfies this requirement.

To summarize, with respect to syntax and comment convention PDL2 and HOL are at least within range of simple mechanical translation.

I. Defaults, Conditional Compilation and Language Restrictions

I1. There will be no defaults in programs which affect the program logic.

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

PDL2 satisfies requirements I1 and I2.

I3. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

I4. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

PDL2 does not satisfy I3 or I4 since there are no compile time statements or variables. It is possible in the ASC implementation to specify the object computer as either the central processor or the peripheral processor. In addition, macros provide some of the same capabilities.

I5. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

Since PDL2 is not an extensible language it does not satisfy the requirement. What is more, it is questionable whether HOL can satisfy the requirement in a meaningful fashion. There is very little extensibility remaining in HOL. The concept of extensibility usually implies something more than defining new data types and infix operators.

I6. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

I7. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

PDL2 at least partially satisfies I6 and I7. Since PDL2 has only been implemented on one machine, these questions have not been fully addressed. They certainly would not effect PDL2 being used as an experimental HOL.

In summary, PDL2 does not have conditional compilation features. With respect to defaults and language restrictions PDL2 and HOL are reasonably close.

J. Efficient Object Representations and Machine Dependencies

J1. The language and its translators will not impose run time costs for unneeded generality. They will be capable of producing efficient code for all programs.

This "requirement" should really be considered to be an objective. Whether or not J1 is satisfied will certainly be controversal since the meaning unneeded or unused generally varies widely and given specific cases can probably be argued forever without reaching agreement. This was certainly a guideline in the design of PDL2 however.

J2. Any optimizations performed by the translator will not change the effect of the program.

PDL2 satisfies this requirement.

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

PDL2 does not allow encapsulated machine language code insertions.

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

In PDL2 the user cannot control the object presentation of data structures other than by knowing the compiler's storage allocation algorithm. The time/space trade-off is made by specifying whether a data structure is packed or not.

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.

PDL2 does not allow open routine. Macros in PDL2 does provide an almost equivalent capability. It is true that scope is different between Macros and

open calls, in practice this difference is usually unimportant.

In summary, with respect to efficient object representation and machine dependencies PDL2 does not have machine language inserts or permit the user to specify object format of data structures.

Other Requirements

Requirements in section K through M are not really technical characteristics. Characteristics such as documentation, support software, and management policies are not really reflected in the language and moreover are important mainly in a production environment.

Summary

There are areas where PDL2 has capabilities which exceed HOL requirements. In particular, the ability to perform operations on array cross-sections. This is merely an extension of allowing operations on entire arrays. To specify an operation on an array cross-section the user simply specifies which elements are to be included in the cross-section. Inclusion of this capability will result in programs being written at a much higher level and in a more readable fashion.

One point which is not clarified in the Tinman is whether or not scope of variables applies across separately compiled modules. PDL2 does have scope across separately compiled modules, and is embedded in a system which determines which procedures or functions need to be recompiled at any time. This permits any procedure or function to be separately compiled.

The "closeness" of PDL2 to HOL is summarized in the table. The major differences are:

- HOL provides for clusters or the equivalent.
- HOL uses a different library approach.
- HOL has extensive compile time features.
- HOL has extensive exception handling.

COMPARATIVE EVALUATION: PDL2 VS. TINMAN

| TINMAN REQUIREMENT | PDL2 RATING |
|---|---|
| **A. DATA AND TYPES** | |
| 1. Typed Language | A |
| 2. Data Types | B |
| 3. Precision | C |
| 4. Fixed Point Numbers | E |
| 5. Character Data | E |
| 6. Arrays | B |
| 7. Records | A |
| **B. OPERATIONS** | |
| 1. Assignment and Reference | C |
| 2. Equivalence | B |
| 3. Relationals | A |
| 4. Arithmetic Operations | C,F-A4 |
| 5. Truncation and Rounding | A |
| 6. Boolean Operations | B |
| 7. Scalar Operations | B |
| 8. Type Conversion | B |
| 9. Changes in Numeric Representation | A |
| 10. I/O Operations | E |
| 11. Power Set Operations | A |
| **C. EXPRESSIONS AND PARAMETERS** | |
| 1. Side Effects | A |
| 2. Operand Structure | A |
| 3. Expressions Permitted | A |
| 4. Constant Expressions | A |
| 5. Consistent Parameter Rules | C |
| 6. Type Agreement in Parameters | B |
| 7. Formal Parameter Kinds | E |
| 8. Formal Parameter Specifications | B |
| 9. Variable Numbers of Parameters | E |
| **D. VARIABLES, LITERALS AND CONSTANTS** | |
| 1. Constant Value Identifiers | A |
| 2. Numeric Literals | A |
| 3. Initial Values of Variables | B |
| 4. Numeric Range and Step Size | D |
| 5. Variable Types | A |
| 6. Pointer Variables | B |
| **E. DEFINITION FACILITIES** | |
| 1. User Definitions Possible | B |
| 2. Consistent Use of Types | A |
| 3. No Default Declarations | A |
| 4. Can Extend Existing Operators | E |
| 5. Type Definitions | E |
| 6. Data Defining Mechanisms | B |
| 7. No Free Union or Subset Types | A |
| 8. Type Initialization | E |

| TINMAN REQUIREMENT | PDL2 RATING |
|---|---|
| **F. SCOPES AND LIBRARIES** | |
| 1. Separate Allocation and Access Allowed | B |
| 2. Limiting Access Scope | E |
| 3. Compile Time Scope Determination | A |
| 4. Libraries Available | F-E5 |
| 5. Library Contents | A |
| 6. Libraires and Compools Indistinguishable | B |
| 7. Standard Library Definition | B |
| **G. CONTROL STRUCTURES** | |
| 1. Kinds of Control Structures | E |
| 2. The Go To | A |
| 3. Conditional Control | B |
| 4. Iterative Control | C |
| 5. Routines | A |
| 6. Parallel Processing | A |
| 7. Exception Handling | E |
| 8. Synchronization and Real Time | F-G1,G7 |
| **H. SYNTAX AND COMMENT CONVENTIONS** | |
| 1. General Characteristics | C |
| 2. No Syntax Extensions | A |
| 3. Source Character Set | A |
| 4. Identifiers and Literals | A |
| 5. Lexical Units and Lines | C |
| 6. Key Words | A |
| 7. Comment Conventions | C |
| 8. Unmatched Parentheses | A |
| 9. Uniform Referent Notation | C |
| 10. Consistency of Meaning | A |
| **I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS** | |
| 1. No Defaults in Program Logic | A |
| 2. Object Representation Specifications Optional | A |
| 3. Compile Time Variables | E |
| 4. Conditional Compilation | E |
| 5. Simple Base Language | E |
| 6. Translator Restrictions | C |
| 7. Object Machine Restrictions | C |
| **J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES** | |
| 1. Efficient Object Code | A |
| 2. Optimizations Do Not Change Program Effect | A |
| 3. Machine Language Insertions | E |
| 4. Object Representation Specifications | E |
| 5. Open and Closed Routine Calls | B |

LEGEND:
A - FULLY SATISFIES
B - EQUIVALENT CAPABILITY IN PDL
C - DIFFERS IN MINOR DETAIL ONLY
D - RESTRICTION NOT IN PDL
E - MISSING FEATURE IN PDL
F - NOT APPLICABLE BECAUSE OF LACK OF OTHER FEATURE

- HOL has encapsulated machine language inserts.

While this may appear to be a large number of differences, comparison with other existing languages will result in a larger list. In addition, PDL2 could be modified to more closely conform to HOL requirements where necessary. In some areas this would be almost trivial. Other areas would require more effort, but modification would definitely be practical.

Therefore while PDL2 in its present form does not satisfy all of the "Tinman" requirements, it is close enough to be quite suitable as interim vehicle with which to zero in on the language defined by the HOL "Tinman."

# An Evaluation of PEARL

J.H. Williams

Computer Science Department

Cornell University

January 1977

This report was prepared for the U.S. Army Electronics Command at Fort Monmouth, New Jersey with the financial support of the Scientific Services Program.

1. Overview of the language

2. Language features

   a) Data types

   b) Data structures

   c) Control structures

   d) Declarations

   e) Program structure

   f) Special or unusual features

3. Language characteristics

   a) Machine independence

   b) Separate compilation

   c) Language efficiency and size

4. The impact of PEARL on the DoD common language effort

5. References

## 1. Overview of the language

All of the information about PEARL which is used in this report was obtained from the PEARL Language Description [ESG-76]. Work on PEARL (Process and Experiment Automation Realtime Language) was begun in 1968. The first description of it was published in 1970. Since that time the language has undergone revisions, has been implemented on eight different process control computers, and has been subsetted for specific application to embedded computer systems with realtime constraints. It is this subset language which is described in [ESG-76] and which is discussed in this report.

The language was designed to provide systems engineers with a high level means of specifying programs for embedded applications. For that reason it emphasizes the areas of input-output specification and realtime processing and avoids many of the more complex and powerful constructs and capabilities of high level programming languages. In this latter, algorithmic aspect it has the appearance of a severely restricted dialect of PL/I.

It is the conclusion of this report that PEARL is not a viable candidate for the DoD common programming language effort, but that the experience gained from its unique approach to input-output specification and realtime control should prove valuable to the forthcoming design efforts.

In this report, we attempt to point out some of the weaknesses which make the language ill-suited for modification or adoption as the DoD common high level algorithmic language. At the same time we hope to encourage its further study by the eventual language design teams.

## 2. Language features

Perhaps the most striking feature of PEARL is its method of factoring a program into two pieces, one hardware dependent and one hardware independent. These are called the system division and the problem division respectively. Within the system division the programmer specifies the particular connections between the input-output devices and their data paths. A rather elaborate formalism is developed for making these specifications, including the ability to use mnemonic names for particular bits, bytes, signals, devices, etc. Then in the problem division the programmer can use these mnemonic identifiers to control the input-output activity. This allows an unusual degree of machine independence in the input-output sections of the program; it is conceivable that one could change the hardware configuration and not have to make any modifications at all to the problem division of a running program.

The language of the problem division of a PEARL program is essentially a PL/I derivative. We describe its features in the remainder of this section, in particular its data-types, data structures, control structures, declarations, program structure, and some of its unique features.

a) Data Types

There are six basic data types with the normal accompanying arithmetic and relational operators.

i) FIXED for integer numbers

ii) FLOAT for floating point numbers

iii) BIT(n) for bit strings

iv) CHAR(n) for character strings

v) CLOCK for time of day

vi) DUR for interval of time

It is possible to denote constants of any of these types, 14:21:06 and 2 HRS 31 MIN being denotations of CLOCK and DUR constants respectively.

In addition, there are five other data types associated with input-output and task control. They differ from the above six types in that there are no operators in the language which manipulate these types. They are:

DEVICE-MODE

FILE-MODE

INTERRUPT-MODE

SIGNAL-MODE

SEMA

More will be said about these types in the discussion on declarations and program structure.

b) Data Structures

It is possible to form collections of the basic data types using either arrays or structures, the former being homogeneous. The array mechanism is essentially that of PL/I with the important difference that all array bounds must be constants. This is true of formal parameter specifications as well, seriously limiting the usefulness of procedures with array parameters. There are no provisions for manipulating sub-arrays or slices. Except for input-output, there are no operators which manipulate entire arrays.

The structure mechanism allows the programmer to create collections of non-homogeneous simple types.

DCL ST STRUCT (N1 FLOAT,N2 BIT(3)) creates a structure called ST which consists of a floating part N1 and a 3-bit string N2 . Components of structures cannot themselves be arrays or structures, nor can a structure have a varying composition as with PASCAL records.

There is no mode declaration facility in PEARL. Basically each STRUCT definition creates a new "mode". However it is possible

to use the LIKE attribute as in PL/I to overcome some of the inconvenience caused by the lack of explicit mode declarations.

c) Control structures

These are essentially PL/I with some restrictions and the addition of a case statement.

i) Goto-statement

It is not permitted to use a GOTO statement to exit a task or procedure.

ii) Conditional-statement

This is the usual IF statement with an optional ELSE and a required delimiter FIN. The conditional expression is of type BIT(1) with '1'B representing true as in PL/I.

iii) Case-statement

This is a generalization of the IF statement wherein the conditional expression is of type integer, and the appropriate statement is executed. There is no provision for case statements with conditional expressions of other data types.

iv) Iteration-statement

This statement is of the form:

```
FOR i FROM a BY b TO c WHILE d
   REPEAT
         stmts
   END;
```

v) Procedures and tasks.

The procedure mechanism is similar to PL/I but with some rather severe restrictions. Parameters can be passed by value or by reference. Neither procedures nor labels may be passed as parameters. The type of the actual parameter must match that of the formal parameter exactly. Thus, procedures cannot be written which will work for arrays of differing sizes. These restrictions imply the need for rather clumsy and inefficient error handling mechanisms in programs. Since procedures can neither exit to indicate abnormal conditions nor call error handling procedures passed to them, some global or parametric flags will need to be raised and interrogated to accomplish error handling. This will lead to run time inefficiences.

The task mechanism is very much simpler than that of PL/I, due to the restrictions on program structure which are discussed below. What remains when one removes the complications of dynamic environments and requires all tasks and procedures to be declared at level zero is a rudimentary but effective mechanism for describing actions associated with real time conditions in the embedded computer system environment. It is possible to schedule tasks AT real times, AFTER particular events or durations of time, WHEN external interrupts occur, ALL some duration UNTIL some clock time, etc. This capability is, of course, greatly enhanced by the basic types, CLOCK and DUR and the ability to manipulate them. Tasks can be suspended, reactivated, and terminated and can be synchronized via the semaphore data types as in many languages.

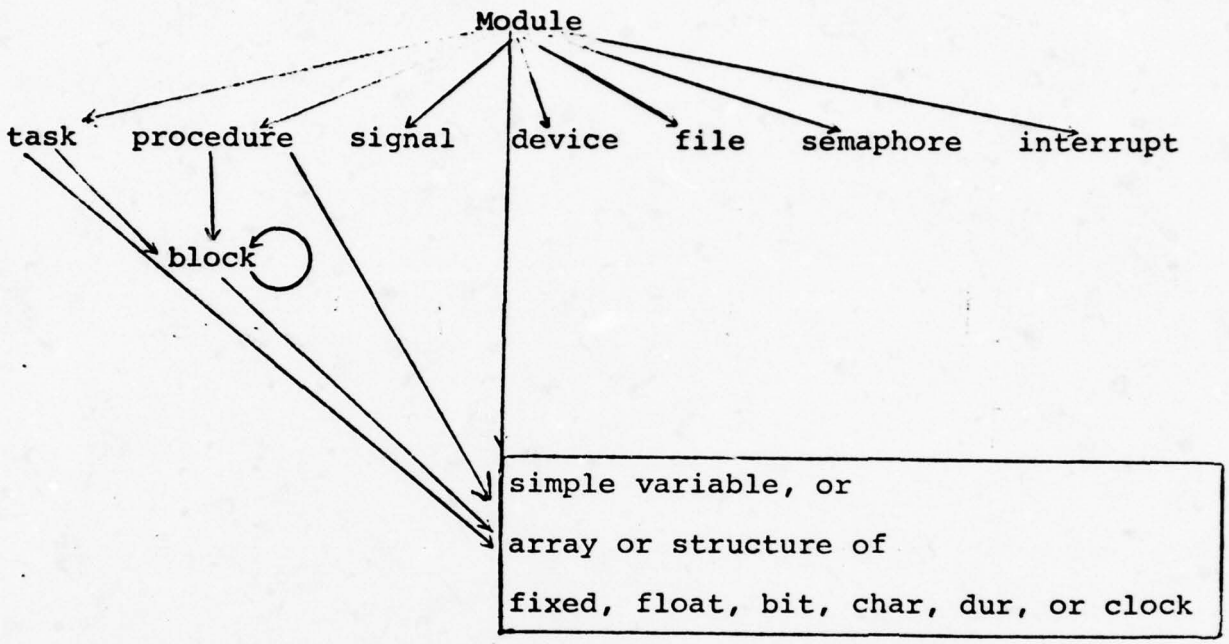
d) Declarations

The declaration mechanism is that of PL/I with some changes. It suffers the same non-orthogonality conditions as does PL/I. In addition, variable names can be of any length, but only the first six characters have any significance. This is an unnecessary concession to implementation shortcuts which can lead to disastrous

consequences in program clarity. There is a modifier to indicate constant declarations. (Curiously a "variable" can be declared INV or invariable!) There is also a mechanism for specifying the attributes of a variable which has been declared in some other program module and is to be considered "common" to this module. In addition, variables can be initialized with the INIT attribute. This initial value is used every time the block is re-entered.

e) Program structure

PEARL programs consist of modules which are separate entities connected only through the common (or SPC for specification) mechanism of their declarations. They can thus be separately compiled. Each module consists of a problem division and a system division as discussed earlier. A module is made up of declarations of variables, tasks and procedures. Tasks and procedures consist of declarations of restricted types of variables and blocks of statements. The only syntactically self-nesting program construct is the block. We illustrate this pictorially in the following diagram. The arrow relationship is read "can contain an instance of."




The most significant restriction of this program structure is the lack of nested procedure definitions.

f) Miscellaneous language features

A variable may be declared "global" and shared with another module. However, any variable so declared must be declared at the module level. Thus if one procedure (or task or nested block within a module) is to be able to share a declaration with some

other module, all tasks, procedures and blocks in that module will also have access to the shared object.

It is possible to specify the length of integer and floating variables. This is done at the module level.

It is not permitted to perform the usual mixed mode arithmetic operations. For example, if A is FLOAT then A+1 is illegal. It is possible to do a few operations of mixed mode, for example, DUR+CLOCK→CLOCK. There are a number of explicit conversion routines, eg A+FLOAT(1) would be legal.

It is possible to declare a procedure to be reentrant "in case several tasks wish to use the procedure simultaneously." The manual gives no other hint of whether procedures can be recursive. This question will be considered more completely in the next section.

3. Language characteristics

a) Machine independence

The language has been designed to be very much machine independent. This is accomplished in large part by the separate system division of program modules. There are other design features which contribute to this independence, such as the provision for declaring the lengths of integers and floating point numbers rather than a mechanism such as DOUBLE for extended precision which is inherently machine dependent. Unfortunately, there are some small machine dependencies built into the language; eg the implicit conversion which takes place during an assignment of the form, FIXED←FLOAT, depends on the wordsize of the machine for its definition. All in all however, the problem division of a PEARL module is highly machine independent, which is rather remarkable for a language designed specifically for realtime control of embedded computer systems.

b) Separate compilation

This was evidently one of the design criteria of PEARL. Because of its simple static storage made possible by the program structure and its independent modules, it is completely separately compilable with almost no work left for the loader.

c) Language efficiency and size

Again, because of the program structure, the limited type-declaration facility, and the compile time typing of all variables, PEARL compilers should produce very efficient object code. As mentioned earlier, the Spartan nature of the language could lead to a somewhat inefficient expression of algorithms at the source level. The language is small enough to allow a compiler to operate efficiently on a small machine. Ease of implementing seems to have been a goal of the language design, sometimes leading to extremes as in the case of variable identifier lengths.

## 4. The impact of PEARL on the DoD common language effort

PEARL does not come close to satisfying the many requirements of the Tinman document [Fi-76]. It would take a great amount of modification and enhancement to make it a viable candidate for the DoD's common language. Indeed, it is essentially a derivative of PL/I with many of the more powerful (and sophisticated) features removed. As such it is not nearly as rich or modern a basic language as, for example, Pascal or Algol-68.

However, the PEARL language experience should have a decided impact on the DoD effort. Since the language was designed specifically for embedded computer systems, the experience gained from its use, particularly the use of its system division, should be valuable to any team attempting to adapt an existing language to meet the Tinman requirements. This language is unique in its approach to specifying input-output hardware in a high order language. This is not to suggest that the actual syntax of PEARL is the best that can be done and should be adopted by the forthcoming design efforts. In fact, at the recent language workshop held at Cornell University, Professor Eberhard Wegner indicated that work is currently underway to radically revise the input-output handling and specification mechanisms of PEARL [Weg-76]. However, the

experience gained from the use of this language and the reasons for the current revisions should prove to be useful input to those design efforts.

## 5. References

[ESG-76]

ESG Elektronik-System-Gesellschaft mb H, PEARL Subset for Avionic Applications, Munich, Germany, June 1976.

[Fis-76]

Fisher, D.A., "A common programming language for the Department of Defense - background and technical requirements", IDA report number P-1191, June 1976.

[Weg-76]

Wegner, E., comments made at the Department of Defense Language Workshop, Cornell University, October 1976.

# DOD-1 Language Evaluation Matrix

Submitted to

the Higher Order Language Working Group

(HOLWG)

January 1977

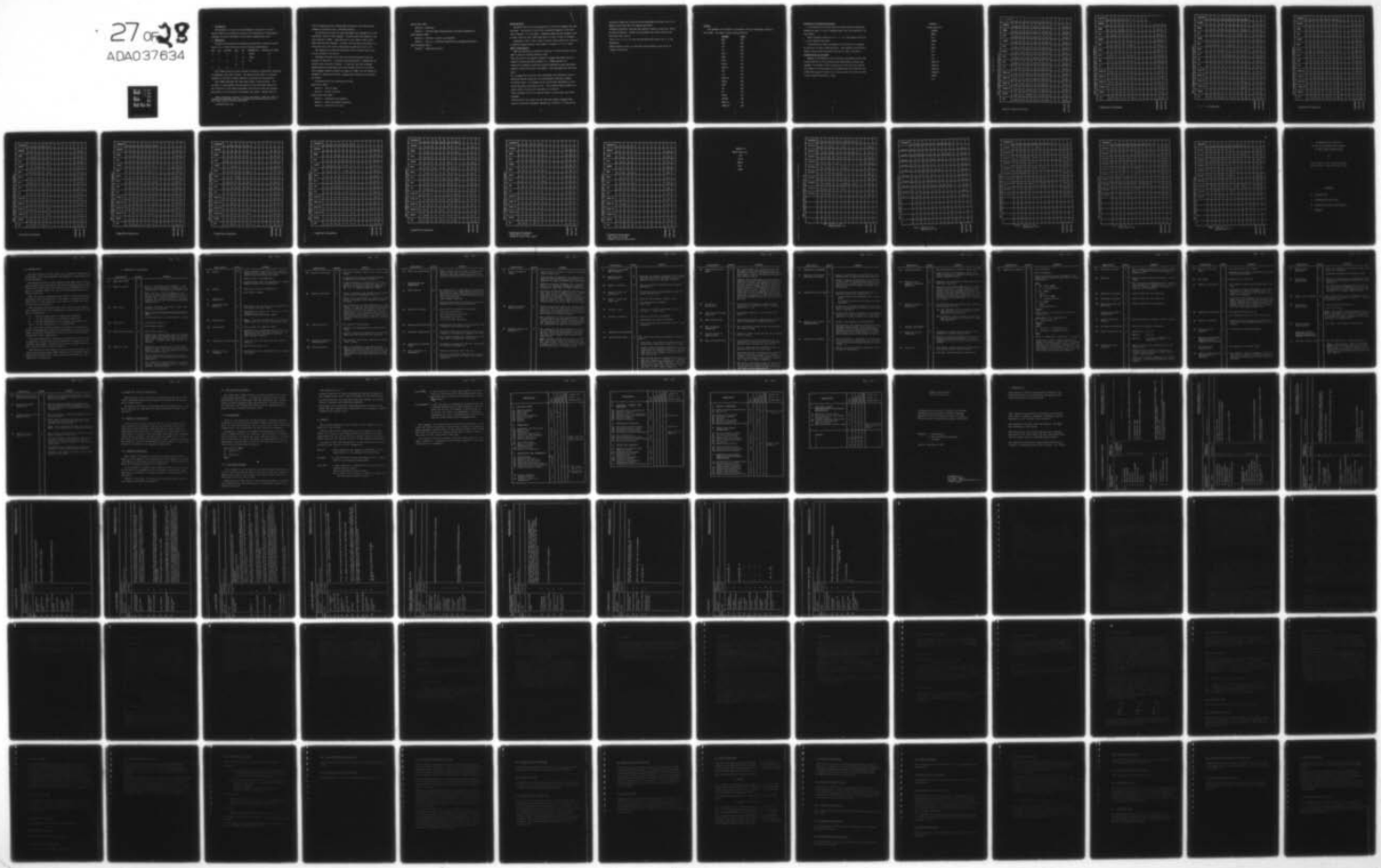
by:

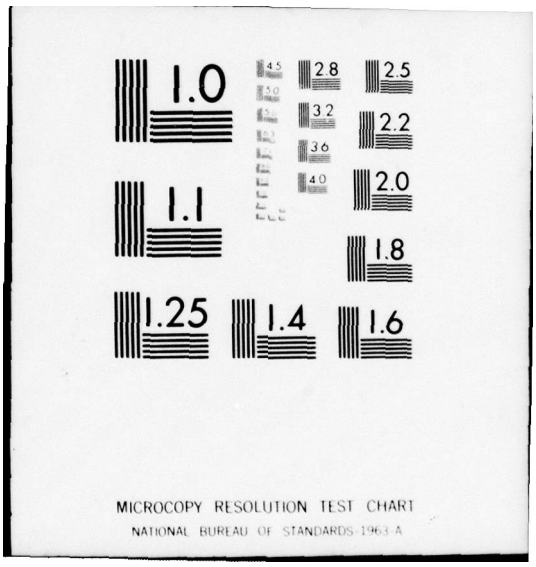

| | |
|---|---|
| Derek S. Morris (Chairman) | CENTACS, Ft Monmouth, NJ |
| Douglas White | RADC, Rome, NY |
| Warren Loper | NELC, San Diego, CA |
| Lloyd Campbell | BRL, Aberdeen, MD |
| Calvin Showalter | NAVAIR, Washington, DC |

AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

27 OF 28
ADA037634




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

## 1. Introduction

This document is a report to the DOD HOLWG on the results of a multi-service attemp to consolidate the results of the evaluation of 23 programming languages by several contractors into an easily readable matrix form.

## 2. Methodology

The matrix was constructed by mapping the contractor's individual scoring system into a unified joint score by the following transformation:

| IBM | SAI+ | SOFTECH+ | RLG+ | CSC | INTERMETRICS* | UNIFIED JOINT SCORE |
|---|---|---|---|---|---|---|
| C | S | S | T | T | Actual | 10 |
| | | | P+ | P+ | Times | 7 |
| P | P | P | P | P | 10 | 5 |
| | | | P- | P- | | 2 |
| F | N | N | F | F | | 0 |

Only TINMAN sections A-J were included in the matrix; applications weightings or management issues were excluded. The reason was that there is no general agreement at the level of detail required to contribute to the evaluation.

Each TINMAN requirement was given equal weight in the evaluation. This was done as it was apparent that the majority of the contractors applied his own "weighting" to the TINMAN requirements implicitly by rating the languages more harshly on some particular requirements than others. Because these im-

+These contractors included a U in their evaluations. Where the symbol U, occured, the requirement was either evaluated from the report text or set at the average value of the other contractors.

* Language vector only.

plicit weightings were not uniform among contractors, the fairest rating scheme was considered to be a uniformed weighting.

The unified joint scores for each requirement were averaged for all the contractual inputs for that language. The points were then summed for each major section of TINMAN. These sums were in turn added to form a total point score for each language. A percent score was then calculated by dividing this total point score by the maximum possible point score (780).

The languages for which the contractors did not provide a point score are not included in this matrix.

Two people from each of the three services participated in the construction of the matrix. Each person selected sections of TINMAN which was closest to his particular interests. In that way, each service member became familiar (to the degree that time allowed) with how all the evaluated languages stacked up against his subset of TINMAN. By this method, we attempted to remove any individual language biases which we as individuals may possess.

The responsibility for rating was as follows:

Derek Morris (ARMY)

Section A. Data and types

Section G. Control Structures

Douglas White (AIR FORCE)

Section C. Expressions and Parameters

Section H. Syntax and Comment Corrections

Section E. Definition Facilities

Warren Loper (NAVY)

Section B. Operations

Section J. Efficient Object Representations and Machine Dependencies

Lloyd Campbell (ARMY)

Section D. Variables, Literals, and Constants

Section I. Defaults, Conditional Compilation and Language Restrictions

Calvin Showalter (NAVY)

Section F. Scopes and Libraries

Resulting Matrix

The matrix was initially constructed for 15 of the 23 languages that were evaluated. The results of that initial construction appears as the front page of Appendix I of this report. Subsequent pages show the average as well as total scores for each TINMAN requirement for each of these 15 languages.

Subsequent to this initial rating individual rating sheets were constructed as material became available. These appear in Appendix II of this report.

Matrix Interpretation

When interpreting the information contained in the matrix below one must keep at least the following points in mind:

*Only the specific (as opposed to general) language requirements were considered in constructing these numbers (i.e., TINMAN sections A-J).

*There was no attempt to weight the relative importance of each requirement relative to each of the other requirements. Each requirement was considered equal.

*If a language did not satisfy some requirement, the difficulty or ease of making the required change was not distinguished in the matrix numbers.

*In certain cases, if a language did not satisfy some requirement it would affect many other requirements as well. This tended to magnify unduely the overall effect of the missing requirement on the score.

*Large languages with lots of features tended to score higher than smaller languages.

*The scoring did not reflect the fact that even though a language might satisfy a particular requirement (perhaps only partially), it might be the

case that a better way of satisfying the requirement entailing a significant amount of work might make the language much better.

*Different evaluators, even when they basically agreed on some point, tended to score differently. SofTech was consistently more severe than say IBM which was rather lenient.

*Some of the languages had only one evaluation while others had up to four evaluations.

*Some evaluations where in a form where the information could not be included in this matrix.

## Findings

The languages can be ordered in accordance with their percentage compliance with TINMAN. The result of that ordering would be:

| Language | Score |
|---|---|
| HALS | 73% |
| CS-4 | 69% |
| LTR | 69% |
| PL/I | 59% |
| RTL/2 | 56% |
| COBOL | 56% |
| CMS-2 | 51% |
| EUCLID | 50% |
| ALGOL 68 | 50% |
| SPL1 | 49% |
| J-73 | 49% |
| CORAL 66 | 47% |
| PASCAL | 46% |
| PEARL | 46% |
| J3B | 45% |
| LIS | 43% |
| TACPOL | 43% |
| FORTRAN | 38% |
| ALGOL 60 | 37% |
| SIMULA 67 | 33% |

Conclusions of the Matrix Evaluation

It is difficult to find any safe conclusions from this information. Perhaps the safest is that no language comes close to satisfying all the requirements.

Modern languages containing more of the real-time aspects called for in TINMAN tended to score well.

The performance of HAL/S and SIMULA 67 in the scoring can somewhat be attributed to their single evaluators. Even though LIS had two evaluations, only one had information that could be used in the matrix.

Recommendations to the HOLWG:

Because of the weakness of the conclusions cited above and the clear non-applicability of this approach toward determining a suitable base language for the DOD-1 effort coupled with the fact that the findings are subject to misinterpretation, we recommend that the information contained herein not be included in any future reports offered by the HOLWG to the interested community at large.

Appendix I

Rating Sheets for:

FORTRAN

COBOL

PL/I

TACPOL

HALS

CMS-2

CS-4

J3B

J73

ALGOL 60

CORAL 66

ALGOL 68

SIMULA 67

PASCAL

LIS

DOD-1 Language Evaluation Score for Sections A-J

| PARAGRAPH | A | B | C | D | E | F | G | H | I | J | | | | Total Pts | Ave Score |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FORTRAN | 23 | 58 | 47 | 25 | 8 | 24 | 20 | 56 | 20 | 14 | | | | 295 | 38 |
| COBOL | 52 | 73 | 32 | 39 | 18 | 45 | 37 | 72 | 38 | 30 | | | | 436 | 56 |
| PL/I | 48 | 84 | 51 | 38 | 17 | 50 | 44 | 67 | 40 | 20 | | | | 459 | 59 |
| TACPOL | 30 | 49 | 33 | 31 | 10 | 43 | 31 | 69 | 20 | 20 | | | | 336 | 43 |
| HALS | 50 | 75 | 70 | 45 | 20 | 70 | 50 | 80 | 65 | 45 | | | | 570 | 73 |
| CMS-2 | 34 | 65 | 56 | 35 | 15 | 37 | 23 | 64 | 39 | 26 | | | | 394 | 51 |
| CS-4 | 42 | 85 | 56 | 47 | 59 | 48 | 53 | 60 | 45 | 43 | | | | 538 | 69 |
| J3B | 29 | 43 | 43 | 31 | 16 | 42 | 23 | 58 | 34 | 35 | | | | 354 | 45 |
| J73 | 31 | 54 | 46 | 25 | 17 | 44 | 28 | 64 | 44 | 26 | | | | 379 | 49 |
| ALGOL 60 | 15 | 40 | 45 | 10 | 25 | 25 | 20 | 65 | 35 | 10 | | | | 290 | 37 |
| CORAL 66 | 36 | 37 | 44 | 22 | 16 | 36 | 25 | 78 | 43 | 31 | | | | 368 | 47 |
| ALGOL 68 | 31 | 60 | 36 | 42 | 53 | 31 | 31 | 66 | 33 | 3 | | | | 386 | 50 |
| SIMULA 67 | 20 | 45 | 25 | 20 | 40 | 40 | 5 | 45 | 10 | 5 | | | | 255 | 33 |
| PASCAL | 28 | 59 | 41 | 33 | 30 | 27 | 27 | 66 | 30 | 16 | | | | 357 | 46 |
| LIS | 15 | 45 | 45 | 35 | 30 | 30 | 25 | 75 | 30 | 5 | | | | 335 | 43 |

* Out of a possible 780 points

DOD-1 Language Evaluation Average Score for Paragraph A

| PARAGRAPH | A1 | A2 | A3 | A4 | A5 | A6 | A7 | | | | | | Evaluators | Total Pts | Ave Score |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FORTRAN | 4 | 7 | 1 | 2 | 1 | 7 | 1 | | | | | | 1 | 23 | 3 |
| COBOL | 7 | 6 | 3 | 10 | 9 | 8 | 9 | | | | | | 2 | 52 | 7 |
| PL/I | 6 | 10 | 8 | 7 | 4* | 7 | 6 | | | | | | 2 | 48 | 7 |
| TACPOL | 7 | 5 | 2 | 7 | 1 | 5 | 3 | | | | | | 3 | 30 | 4 |
| HALS | 10 | 5 | 10 | 5 | 0 | 10 | 10 | | | | | | 1 | 50 | 7 |
| CMS-2 | 5 | 7 | 0 | 9 | 1 | 5 | 7 | | | | | | 2 | 34 | 5 |
| CS-4 | 10 | 7 | 7 | 1 | 1 | 8 | 8* | | | | | | 4 | 42 | 6 |
| J3B | 7 | 7 | 1 | 3 | 0 | 6 | 5 | | | | | | 2 | 29 | 4 |
| J73 | 5 | 6 | 5 | 1 | 3 | 6 | 5 | | | | | | 2 | 31 | 4 |
| ALGOL 60 | 10 | 0 | 0 | 0 | 0 | 5 | 0 | | | | | | 1 | 15 | 2 |
| CORAL 66 | 7 | 6 | 3 | 9 | 0 | 6 | 5 | | | | | | 2 | 36 | 5 |
| ALGOL 68 | 10 | 3 | 3 | 0 | 3 | 5 | 5 | | | | | | 2 | 31 | 4 |
| SIMULA 67 | 5 | 5 | 0 | 0 | 0 | 5 | 5 | | | | | | 1 | 20 | 3 |
| PASCAL | 7 | 5 | 0 | 0 | 4 | 6 | 6 | | | | | | 2 | 28 | 4 |
| LIS | 10 | 0 | 0 | 0 | 5 | 0 | 10 | | | | | | 1 | 15 | 4 |

* Significant Disagreement

DOD-1 Language Evaluation Average Score for Paragraph B

| PARAGRAPH | B1 | B2 | B3 | B4 | B5 | B6 | B7 | B8 | B9 | B10 | B11 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FORTRAN | 3 | 4 | 7 | 9 | 8 | 7 | 0 | 5 | 3 | 5 | 7 | | 2 | 58 | 5 |
| COBOL | 9 | 8 | 9 | 10 | 8 | 6 | 6* | 0 | 7 | 10 | 0 | | 2 | 73 | 7 |
| PL/I | 9 | 9 | 10 | 10 | 4 | 8 | 8 | 4 | 10 | 7 | 5 | | 2 | 84 | 8 |
| TACPOL | 5 | 5 | 6 | 6 | 2 | 6 | 2 | 5 | 5 | 5 | 2 | | 3 | 49 | 5 |
| HALS | 10 | 10 | 10 | 5 | 5 | 10 | 10 | 5 | 5 | 5 | 0 | | 1 | 75 | 7 |
| CMS-2 | 8 | 6 | 6 | 10 | 6 | 6 | 5 | 5 | 4* | 4* | 5* | | 2 | 65 | 6 |
| CS-4 | 9 | 7 | 9 | 7 | 10 | 10 | 5* | 6 | 9 | 4* | 9 | | 4 | 85 | 8 |
| J3B | 5 | 8 | 4 | 6 | 3 | 6 | 3 | 3 | 5 | 0 | 0 | | 2 | 43 | 4 |
| J73 | 6 | 6 | 8 | 8 | 6 | 7 | 3 | 4 | 2 | 0 | 4 | | 2 | 54 | 5 |
| ALGOL 60 | 5 | 0 | 10 | 5 | 10 | 5 | 0 | 5 | 0 | 0 | 0 | | 1 | 40 | 4 |
| CORAL 66 | 8 | 8 | 4 | 6 | 0 | 5 | 0 | 4 | 1 | 0 | 1 | | 2 | 37 | 4 |
| ALGOL 68 | 8 | 5 | 8 | 5 | 8 | 5 | 8 | 3 | 0 | 10 | 0 | | 2 | 60 | 4 |
| SIMULA 67 | 5 | 5 | 5 | 10 | 5 | 5 | 0 | 0 | 0 | 10 | 0 | | 1 | 45 | 4 |
| PASCAL | 6 | 6 | 6 | 6 | 5 | 3 | 8 | 0 | 8 | 6 | 5 | | 2 | 59 | 5 |
| LIS | 10 | 0 | 10 | 10 | 0 | 5 | 10 | 0 | 0 | 0 | 0 | | 1 | 45 | 4 |
| | | | | | | | | | | | | | Evaluators | Total Pts | Ave Score |

* Significant Disagreement

DOD-1 Language Evaluation Average Score for Paragraph C

| PARAGRAPH | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FORTRAN | 5* | 8 | 8 | 5 | 10 | 8 | 3 | 0 | 0 | | | | 2 | 47 | 5 |
| COBOL | 9 | 9 | 0 | 0 | 8 | 0 | 1 | 5* | 0 | | | | 2 | 32 | 4 |
| PL/I | 5* | 5 | 10 | 8 | 8 | 5 | 5 | 5 | 0 | | | | 2 | 51 | 6 |
| TACPOL | 2 | 5 | 5 | 2 | 3 | 7 | 5 | 2 | 2 | | | | 3 | 33 | 4 |
| HALS | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 0 | 0 | | | | 1 | 70 | 8 |
| CMS-2 | 10 | 6 | 9 | 10 | 8 | 5 | 5 | 0 | 3 | | | | 2 | 56 | 6 |
| CS-4 | 8 | 8 | 9 | 8 | 4 | 9 | 4 | 5 | 1 | | | | 4 | 56 | 6 |
| J3B | 5* | 8 | 10 | 3 | 5 | 6 | 5 | 0 | 1 | | | | 2 | 43 | 5 |
| J73 | 0 | 6 | 10 | 8 | 5 | 5 | 5 | 4 | 3 | | | | 2 | 46 | 5 |
| ALGOL 60 | 10 | 5 | 10 | 0 | 5 | 10 | 5 | 0 | 0 | | | | 1 | 45 | 5 |
| CORAL 66 | 4 | 5 | 10 | 5* | 5 | 5 | 6 | 4 | 0 | | | | 2 | 44 | 5 |
| ALGOL 68 | 0 | 3 | 10 | 0 | 5 | 5 | 5 | 0 | 8 | | | | 2 | 36 | 4 |
| SIMULA 67 | 0 | 0 | 10 | 0 | 5 | 5 | 0 | 5 | 0 | | | | 1 | 25 | 3 |
| PASCAL | 5* | 8 | 10 | 0 | 8 | 5 | 4* | 1 | 0 | | | | 2 | 41 | 5 |
| LIS | 0 | 10 | 10 | 5 | 5 | 10 | 5 | 0 | 0 | | | | 1 | 45 | 5 |
| | | | | | | | | | | | | | Evaluators | Total Pts | Ave Score |

* Significant Disagreement

DOD-1 Language Evaluation Average Score for Paragraph D

| PARAGRAPH | D1 | D2 | D3 | D4 | D5 | D6 | | | | | | | Evaluators | Total Pts | Ave Score |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FORTRAN | 5* | 9 | 9 | 0 | 2 | 0 | | | | | | | 2 | 25 | 4 |
| COBOL | 5* | 10 | 7* | 9 | 8 | 0 | | | | | | | 2 | 39 | 7 |
| PL/I | 3 | 9 | 9 | 5 | 7 | 5 | | | | | | | 2 | 38 | 6 |
| TACPOL | 9 | 8 | 3* | 6 | 5 | 0 | | | | | | | 3 | 31 | 5 |
| HALS | 10 | 10 | 10 | 0 | 5 | 10 | | | | | | | 1 | 45 | 8 |
| CMS-2 | 10 | 5 | 10 | 7 | 2 | 1 | | | | | | | 2 | 35 | 6 |
| CS-4 | 10 | 9 | 9 | 7 | 9 | 3* | | | | | | | 4 | 47 | 8 |
| J3B | 10 | 5 | 6 | 1 | 3 | 6 | | | | | | | 2 | 31 | 5 |
| J73 | 7* | 6 | 5 | 1 | 4* | 2 | | | | | | | 2 | 25 | 4 |
| ALGOL 60 | 0 | 5 | 0 | 0 | 5 | 0 | | | | | | | 1 | 10 | 2 |
| CORAL 66 | 5* | 5 | 7 | 0 | 1 | 4 | | | | | | | 2 | 22 | 4 |
| ALGOL 68 | 10 | 5 | 9 | 0 | 8 | 10 | | | | | | | 2 | 42 | 7 |
| SIMULA 67 | 0 | 5 | 0 | 0 | 5 | 10 | | | | | | | 1 | 20 | 3 |
| PASCAL | 10 | 8 | 0 | 1 | 10 | 4* | | | | | | | 2 | 33 | 6 |
| LIS | 10 | 10 | 5 | 0 | 10 | 0 | | | | | | | 1 | 35 | 6 |

* Significant Disagreement

DOD-1 Language Evaluation Average Score for Paragraph E

| PARAGRAPH | E1 | E2 | E3 | E4 | E5 | E6 | E7 | E8 | | | | | Evaluators | Total Pts | Ave Score |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FORTRAN | 0 | 0 | 5 | 0 | 0 | 0 | 3 | 0 | | | | | 2 | 8 | 1 |
| COBOL | 3 | 0 | 5* | 0 | 0 | 2 | 8 | 0 | | | | | 2 | 18 | 2 |
| PL/I | 3 | 0 | 3 | 0 | 0 | 3 | 8 | 0 | | | | | 2 | 17 | 2 |
| TACPOL | 0 | 0 | 5 | 0 | 0 | 2 | 3 | 0 | | | | | 3 | 10 | 1 |
| HALS | 5 | 0 | 5 | 0 | 0 | 0 | 10 | 0 | | | | | 1 | 20 | 3 |
| CMS-2 | 3 | 0 | 5 | 1 | 0 | 5 | 1 | 0 | | | | | 2 | 15 | 2 |
| CS-4 | 5 | 5* | 10 | 3* | 8 | 10 | 8 | 10 | | | | | 4 | 59 | 7 |
| J3B | 0 | 0 | 10 | 0 | 3 | 3 | 0 | 0 | | | | | 2 | 16 | 2 |
| J73 | 1 | 0 | 8 | 0 | 0 | 4 | 4 | 0 | | | | | 2 | 17 | 2 |
| ALGOL 60 | 0 | 0 | 10 | 0 | 0 | 5 | 10 | 0 | | | | | 1 | 25 | 3 |
| CORAL 66 | 0 | 0 | 10 | 0 | 0 | 3 | 3 | 0 | | | | | 2 | 16 | 2 |
| ALGOL 68 | 10 | 10 | 10 | 8 | 0 | 5 | 10 | 0 | | | | | 2 | 53 | 7 |
| SIMULA 67 | 5 | 5 | 5 | 0 | 5 | 5 | 10 | 5 | | | | | 1 | 40 | 5 |
| PASCAL | 6 | 5* | 9 | 3 | 5 | 9 | 1 | 1 | | | | | 2 | 39 | 5 |
| LIS | 5 | 10 | 10 | 0 | 0 | 5 | 0 | 0 | | | | | 1 | 30 | 4 |

* Significant Disagreement

DOD-1 Language Evaluation Average Score for Paragraph F

| PARAGRAPH | F1 | F2 | F3 | F4 | F5 | F6 | F7 | | | | | | Evaluators | Total Pts | Ave Score |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FORTRAN | 3 | 0 | 8 | 8 | 0 | 0 | 5 | | | | | | 2 | 24 | 3 |
| COBOL | 1 | 3 | 7* | 8 | 9 | 10 | 7 | | | | | | 2 | 45 | 0 |
| PL/I | 10 | 4 | 9 | 9 | 5* | 5* | 8 | | | | | | 2 | 50 | 7 |
| TACPOL | 5 | 3 | 10 | 5 | 6 | 7 | 7 | | | | | | 3 | 43 | 6 |
| HALS | 10 | 10 | 10 | 10 | 10 | 10 | 10 | | | | | | 1 | 70 | 10 |
| CMS-2 | 3 | 1 | 9 | 10 | 5 | 6 | 3 | | | | | | 2 | 37 | 5 |
| CS-4 | 9 | 10 | 10 | 4 | 5 | 5 | 5* | | | | | | 4 | 48 | 7 |
| J3B | 8 | 4 | 10 | 5* | 6 | 4* | 5* | | | | | | 2 | 42 | 6 |
| J73 | 6 | 3 | 10 | 7 | 9 | 8 | 1 | | | | | | 2 | 44 | 6 |
| ALGOL 60 | 10 | 0 | 10 | 1 | 1 | 1 | 0 | | | | | | 1 | 25 | 5 |
| CORAL 66 | 8 | 1 | 10 | 8 | 5 | 4 | 0 | | | | | | 2 | 36 | 5 |
| ALGOL 68 | 8 | 3 | 10 | 0 | 0 | 0 | 10 | | | | | | 2 | 31 | 4 |
| SIMULA 67 | 5 | 5 | 10 | 5 | 0 | 5 | 10 | | | | | | 1 | 40 | 6 |
| PASCAL | 3 | 3 | 10 | 3 | 0 | 0 | 8 | | | | | | 2 | 27 | 4 |
| LIS | 5 | 5 | 10 | 1 | 1 | 0 | 10 | | | | | | 1 | 30 | 4 |

* Significant Disagreement

DOD-1 Language Evaluation Average Score for Paragraph G

| PARAGRAPH | G1 | G2 | G3 | G4 | G5 | G6 | G7 | G8 | | | | | Evaluators | Total Pts | Ave Score |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FORTRAN | 5 | 8 | 2 | 5 | 0 | 0 | 0 | 0 | | | | | 2 | 20 | 3 |
| COBOL | 7 | 8 | 7 | 7* | 0 | 0 | 8 | 0 | | | | | 2 | 37 | 5 |
| PL/I | 7 | 5 | 6 | 7 | 10 | 0 | 9 | 0 | | | | | 2 | 44 | 6 |
| TACPOL | 6 | 6* | 5* | 6 | 3 | 0 | 4 | 1 | | | | | 3 | 31 | 4 |
| HALS | 5 | 10 | 5 | 5 | 0 | 10 | 5 | 10 | | | | | 1 | 50 | 6 |
| CMS-2 | 5 | 5 | 6 | 6 | 0 | 0 | 1 | 0 | | | | | 2 | 23 | 3 |
| CS-4 | 7 | 7 | 8 | 8 | 0 | 7 | 9 | 7 | | | | | 4 | 53 | 7 |
| J3B | 4 | 6 | 4* | 4* | 5* | 0 | 0 | 0 | | | | | 2 | 23 | 3 |
| J73 | 6 | 3 | 7 | 6 | 4 | 0 | 2 | 0 | | | | | 2 | 28 | 4 |
| ALGOL 60 | 5 | 0 | 5 | 5 | 5 | 0 | 0 | 0 | | | | | 1 | 20 | 3 |
| CORAL 66 | 5 | 3 | 6 | 6 | 5 | 0 | 0 | 0 | | | | | 2 | 25 | 3 |
| ALGOL 68 | 5 | 0 | 5 | 5 | 5 | 8 | 3 | 0 | | | | | 2 | 31 | 4 |
| SIMULA 67 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | | | | | 1 | 5 | 1 |
| PASCAL | 5 | 3 | 5 | 3 | 5 | 0 | 3 | 3 | | | | | 2 | 27 | 3 |
| LIS | 5 | 0 | 5 | 5 | 5 | 5 | 0 | 0 | | | | | 1 | 25 | 3 |

* Significant Disagreement

DOD-1 Language Evaluation Average Score for Paragraph H

| PARAGRAPH | H1 | H2 | H3 | H4 | H5 | H6 | H7 | H8 | H9 | H10 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FORTRAN | 5 | 10 | 10 | 5 | 0 | 3 | 3 | 10 | 5 | 5* | | | 2 | 56 | 6 |
| COBOL | 6 | 10 | 10 | 7 | 2 | 5 | 7 | 10 | 5* | 10 | | | 2 | 72 | 7 |
| PL/I | 5 | 10 | 10 | 5 | 5 | 8 | 8 | 3 | 8 | 5 | | | 2 | 67 | 7 |
| TACPOL | 7 | 10 | 10 | 7 | 3 | 7* | 5 | 10 | 7 | 3 | | | 3 | 69 | 7 |
| HALS | 10 | 10 | 10 | 10 | 5 | 10 | 5 | 10 | 10 | 0 | | | 1 | 80 | 8 |
| CMS-2 | 5 | 10 | 10 | 8 | 0 | 5 | 5 | 10 | 6 | 5 | | | 2 | 64 | 7 |
| CS-4 | 6 | 9 | 5* | 8 | 5* | 5* | 5* | 10 | 7 | 10 | | | 4 | 60 | 7 |
| J3B | 6 | 10 | 5* | 5 | 0 | 8 | 5 | 10 | 6 | 3 | | | 2 | 58 | 6 |
| J73 | 6 | 10 | 10 | 6 | 3 | 8 | 6 | 10 | 0 | 5 | | | 2 | 64 | 6 |
| ALGOL 60 | 10 | 10 | 10 | 5 | 0 | 5 | 5 | 10 | 0 | 10 | | | 1 | 65 | 7 |
| CORAL 66 | 6 | 8 | 10 | 8 | 5* | 10 | 6 | 10 | 5* | 10 | | | 2 | 78 | 8 |
| ALGOL 68 | 8 | 3 | 10 | 8 | 3 | 8 | 5 | 10 | 3 | 8 | | | 2 | 66 | 7 |
| SIMULA 67 | 10 | 10 | 0 | 5 | 0 | 10 | 0 | 10 | 0 | 0 | | | 1 | 45 | 5 |
| PASCAL | 6 | 10 | 5 | 6 | 0 | 10 | 6 | 10 | 5* | 8 | | | 2 | 66 | 7 |
| LIS | 10 | 10 | 0 | 5 | 5 | 10 | 10 | 10 | 10 | 5 | | | 1 | 75 | 8 |
| | | | | | | | | | | | | | Evaluators | Total Pts | Ave Score |

* Significant Disagreement

DOD-1 Language Evaluation Average Score for Paragraph I

| PARAGRAPH | I1 | I2 | I3 | I4 | I5 | I6 | I7 | | | | | | Evaluators | Total Pts | Ave Score |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FORTRAN | 0 | 0 | 0 | 0 | 7 | 5 | 8 | | | | | | 2 | 20 | 3 |
| COBOL | 10 | 8 | 5* | 0 | 1 | 4* | 10 | | | | | | 2 | 38 | 5 |
| PL/I | 3 | 9 | 5* | 5* | 3 | 5* | 10 | | | | | | 2 | 40 | 6 |
| TACPOL | 2 | 4 | 0 | 0 | 0 | 5 | 9 | | | | | | 3 | 20 | 3 |
| HALS | 10 | 10 | 10 | 5 | 10 | 10 | 10 | | | | | | 1 | 65 | 9 |
| CMS-2 | 10 | 8 | 2 | 1 | 0 | 8 | 10 | | | | | | 2 | 39 | 6 |
| CS-4 | 8 | 7 | 7 | 2 | 5 | 6 | 10 | | | | | | 4 | 45 | 6 |
| J3B | 3 | 8 | 0 | 7* | 6 | 5 | 5* | | | | | | 2 | 34 | 5 |
| J73 | 3 | 5 | 9 | 6 | 9 | 3 | 9 | | | | | | 2 | 44 | 6 |
| ALGOL 60 | 10 | 5 | 0 | 0 | 10 | 0 | 10 | | | | | | 1 | 35 | 5 |
| CORAL 66 | 8 | 8 | 0 | 0 | 10 | 7 | 10 | | | | | | 2 | 43 | 6 |
| ALGOL 68 | 10 | 5 | 4 | 0 | 5 | 0 | 9 | | | | | | 2 | 33 | 5 |
| SIMULA 67 | 0 | 0 | 0 | 0 | 5 | 0 | 5 | | | | | | 1 | 10 | 1 |
| PASCAL | 5* | 5 | 1 | 0 | 10 | 0 | 9 | | | | | | 2 | 30 | 4 |
| LIS | 10 | 10 | 0 | -' | 10 | -' | -' | | | | | | 1 | 30 | 8 |

* Significant Disagreement
' Not counted in average
counted as zero in total points

DOD-1 Language Evaluation Average Score for Paragraph J

| PARAGRAPH | J1 | J2 | J3 | J4 | J5 | | | | | | | | Evaluators | Total Pts | Ave Score |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FORTRAN | 10 | 4 | 0 | 0 | 0 | | | | | | | | 2 | 14 | 3 |
| COBOL | 6 | 10 | 6 | 8 | 0 | | | | | | | | 2 | 30 | 6 |
| PL/I | 5 | 5* | 3 | 4* | 3 | | | | | | | | 2 | 20 | 4 |
| TACPOL | 7 | 0 | 6 | 5 | 2 | | | | | | | | 3 | 20 | 4 |
| HALS | 10 | 10 | 5 | 10 | 10 | | | | | | | | 1 | 45 | 9 |
| CMS-2 | -' | 10 | 10 | 6 | 0 | | | | | | | | 2 | 26 | 6 |
| CS-4 | 7 | 10 | 10 | 6* | 10 | | | | | | | | 4 | 43 | 9 |
| J3B | 8 | 8 | 4* | 5 | 10 | | | | | | | | 2 | 35 | 7 |
| J73 | 7 | 5* | 3 | 7 | 4* | | | | | | | | 2 | 26 | 5 |
| ALGOL 60 | -' | 10 | 0 | 0 | 0 | | | | | | | | 1 | 10 | 3 |
| CORAL 66 | 8 | 10 | 5* | 5 | 3 | | | | | | | | 2 | 31 | 6 |
| ALGOL 68 | 0 | 3 | 0 | 0 | 0 | | | | | | | | 2 | 3 | 1 |
| SIMULA 67 | 0 | 5 | 0 | 0 | 0 | | | | | | | | 1 | 5 | 1 |
| PASCAL | 5 | 8 | 3 | 0 | 0 | | | | | | | | 2 | 16 | 3 |
| LIS | -' | -' | 0 | 5 | 0 | | | | | | | | 1 | 5 | 1 |

* Significant Disagreement

' Not counted in average
counted as zero in total points

Appendix II

Rating Sheets for:

LTR

RTL/2

EUCLID

SPL1

PEARL

DOD-1 Language Evaluation Average Score for LTR

| PARAGRAPH | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | | Evaluators | Total Pts | Ave Score |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SECTION A | 10 | 10 | 0 | 10 | 2 | 6 | 10 | - | - | - | - | | 1 | 48 | 7 |
| SECTION B | 8 | 10 | 8 | 10 | 8 | 10 | 4 | 6 | 10 | 8 | 0 | | 1 | 82 | 7 |
| SECTION C | 10 | 10 | 10 | 0 | 8 | 10 | 8 | 4 | 0 | - | - | | 1 | 60 | 7 |
| SECTION D | 0 | 10 | 10 | 8 | 4 | 6 | - | - | - | - | - | | 1 | 38 | 6 |
| SECTION E | 0 | 0 | 10 | 0 | 0 | 0 | 10 | 0 | - | - | - | | 1 | 20 | 3 |
| SECTION F | 8 | 8 | 10 | 8 | 10 | 10 | 6 | - | - | - | - | | 1 | 60 | 9 |
| SECTION G | 10 | 10 | 8 | 8 | 0 | 10 | 8 | 10 | - | - | - | | 1 | 64 | 8 |
| SECTION H | 8 | 10 | 10 | 6 | 0 | 8 | 10 | 10 | 6 | 10 | - | | 1 | 78 | 8 |
| SECTION I | 10 | 10 | 0 | 8 | 8 | 8 | 10 | - | - | - | - | | 1 | 54 | 8 |
| SECTION J | 8 | 10 | 10 | 8 | 0 | - | - | - | - | - | - | | 1 | 36 | 7 |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |

LTR: Total Pts = 540
Percent Score = 69

DOD-1 Language Evaluation Average Score for RTL/2

| PARAGRAPH | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | | Evaluators | Total Pts | Ave Score |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SECTION A | 10 | 5 | 0 | 5 | 5 | 5 | 0 | - | - | - | - | | 1 | 30 | 4 |
| SECTION B | 10 | 10 | 10 | 5 | 10 | 5 | 0 | 5 | 10 | 10 | 5 | | 1 | 80 | 7 |
| SECTION C | 5 | 5 | 5 | 0 | 10 | 10 | 5 | 0 | 0 | - | - | | 1 | 40 | 4 |
| SECTION D | 10 | 5 | 5 | 0 | 5 | 5 | - | - | - | - | - | | 1 | 30 | 5 |
| SECTION E | 5 | 0 | 10 | 0 | 10 | 5 | 10 | 0 | - | - | - | | 1 | 40 | 5 |
| SECTION F | 0 | 5 | 10 | 5 | 5 | 5 | 5 | - | - | - | - | | 1 | 35 | 5 |
| SECTION G | 5 | 5 | 5 | 5 | 10 | 5 | 5 | 5 | - | - | - | | 1 | 45 | 6 |
| SECTION H | 10 | 10 | 10 | 5 | 10 | 10 | 5 | 10 | 10 | 10 | - | | 1 | 90 | 9 |
| SECTION I | 5 | 5 | 5 | 0 | - | 5 | 5 | - | - | - | - | | 1 | 25 | 4 |
| SECTION J | 5 | 10 | 5 | 0 | - | - | - | - | - | - | - | | 1 | 20 | 5 |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |

RTL/2: Total Pts = 435
Percent Score = 56

DOD-1 Language Evaluation Average Score for EUCLID

| PARAGRAPH | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | | Evaluators | Total Pts | Ave Score |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SECTION A | 10 | 5 | 0 | 0 | 0 | 7 | 10 | - | - | - | - | | 1 | 32 | 5 |
| SECTION B | 7 | 7 | 5 | 5 | 5 | 5 | 0 | 5 | 5 | 0 | 7 | | 1 | 51 | 5 |
| SECTION C | 0 | 7 | 10 | 10 | 5 | 10 | 3 | 0 | 0 | - | - | | 1 | 45 | 5 |
| SECTION D | 10 | 5 | 10 | 5 | 10 | 5 | - | - | - | - | - | | 1 | 45 | 8 |
| SECTION E | 5 | 10 | 7 | 0 | 5 | 10 | - | 10 | - | - | - | | 1 | 47 | 6 |
| SECTION F | 10 | 10 | 10 | 0 | 0 | 0 | 0 | - | - | - | - | | 1 | 30 | 4 |
| SECTION G | 5 | 0 | 7 | 7 | 5 | 0 | 0 | 0 | - | - | - | | 1 | 24 | 3 |
| SECTION H | 7 | 10 | 0 | 5 | 10 | 0 | 7 | 10 | 7 | 5 | - | | 1 | 61 | 6 |
| SECTION I | 5 | 7 | 0 | 0 | 10 | 5 | 7 | - | - | - | - | | 1 | 34 | 5 |
| SECTION J | - | - | 7 | 7 | 5 | - | - | - | - | - | - | | 1 | 19 | 6 |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |

EUCLID: Total Pts = 388
Percent Score = 50

DOD-1 Language Evaluation Average Score for SPL1

| PARAGRAPH | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | | Evaluators | Total Pts | Ave Score |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SECTION A | 10 | 7 | 0 | 0 | 0 | 5 | 0 | - | - | - | - | | 1 | 22 | 3 |
| SECTION B | 7 | 7 | 7 | 7 | - | 7 | 10 | 5 | 3 | 5 | 5 | | 1 | 63 | 6 |
| SECTION C | 0 | 7 | 10 | 7 | 10 | 7 | 5 | 0 | 3 | - | - | | 1 | 49 | 5 |
| SECTION D | 0 | 7 | 7 | 0 | 7 | 5 | - | - | - | - | - | | 1 | 26 | 4 |
| SECTION E | 3 | 0 | 10 | 0 | 3 | 5 | 10 | 3 | - | - | - | | 1 | 34 | 4 |
| SECTION F | 5 | 5 | 10 | 0 | 3 | - | 0 | - | - | - | - | | 1 | 23 | 3 |
| SECTION G | 7 | 10 | 7 | 5 | 0 | 5 | 0 | 3 | - | - | - | | 1 | 47 | 6 |
| SECTION H | 10 | 10 | 10 | 10 | 10 | 10 | 7 | 10 | 10 | 10 | - | | 1 | 97 | 10 |
| SECTION I | 0 | 7 | 0 | 0 | 0 | 0 | 10 | - | - | - | - | | 1 | 17 | 2 |
| SECTION J | 5 | - | 0 | 0 | 0 | - | - | - | - | - | - | | 1 | 5 | 1 |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |

SPL1: Total Pts = 383
Percent Score = 49

DOD-1 Language Evaluation Average Score for PEARL

| PARAGRAPH | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SECTION A | 10 | 5 | 5 | 0 | 0 | 5 | 0 | - | - | - | - | | 1 | 25 | 3 |
| SECTION B | 5 | 7 | 5 | 7 | - | 5 | 0 | 5 | 0 | 10 | 0 | | 1 | 44 | 4 |
| SECTION C | 0 | 10 | 10 | 0 | 10 | 7 | 5 | 0 | 0 | - | - | | 1 | 42 | 5 |
| SECTION D | 10 | 3 | 7 | 0 | 0 | 0 | - | - | - | - | - | | 1 | 20 | 3 |
| SECTION E | 0 | 0 | 7 | 0 | 0 | 0 | 10 | 0 | - | - | - | | 1 | 17 | 2 |
| SECTION F | 10 | 3 | 10 | 10 | 5 | 10 | 10 | - | - | - | - | | 1 | 58 | 8 |
| SECTION G | 7 | 7 | 5 | 7 | 0 | 5 | 3 | 7 | - | - | - | | 1 | 41 | 6 |
| SECTION H | 7 | 10 | 5 | 5 | 0 | 5 | 7 | 10 | 7 | 10 | - | | 1 | 66 | 7 |
| SECTION I | 5 | 5 | 10 | 0 | - | 5 | 10 | - | - | - | - | | 1 | 35 | 6 |
| SECTION J | 5 | - | 5 | 0 | 0 | - | - | - | - | - | - | | 1 | 10 | 3 |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |

PEARL: Total Pts = 358
Percent Score = 46

Evaluators | Total Pts | Ave Score

# A COMPARATIVE EVALUATION OF THE LIS LANGUAGE WITH RESPECT TO THE TINMAN REQUIREMENTS

by

J.D. Ichbiah, CII - Honeywell Bull
Louveciennes, France, October 1976

CONTENTS

0. INTRODUCTION

1. COMPARATIVE EVALUATION

2. BEYOND THE TINMAN REQUIREMENTS

3. SUMMARY

## 0. INTRODUCTION

The main object of this report is to present a comparative evaluation of the LIS language with respect to the Tinman set of requirements.

The second section of this report lists some areas of language design where LIS offers facilities which go beyond what is strictly required by the Tinman and which should be retained in a language aiming for the best achievable within the state of the art.

The last section summarizes the results of the comparative evaluation and, in particular, the degree of easiness (resp. difficulty) of the extensions that would be needed in order to meet the requirements.

The following conventions have been used for the comparative evaluation. For each requirement we have indicated a degree of compliance :

T - If the requirement is completely satisfied
P - If the requirement is partially satisfied
F - If the language fails to satisfy the requirement
N.A. - If the requirement is not applicable.

In a few cases we have used T- to indicate a requirement which is fundamentally satisfied with minor or easily repairable exceptions. Similarly T+ has been used where LIS goes beyond the strict requirement.

For each requirement satisfied an explanation of how this is achieved is given. For requirements not, or partially satisfied, the comment evaluates the easiness and/or desirability and difficulty of the extension.

References have been made to the LIS reference manual and also to my previous (May 1976) comments of the Tinman report. Additional comments have also been provided along with the comparative evaluation.

## 1. COMPARATIVE EVALUATION

| | REQUIREMENT | DEGREE | COMMENT |
|---|---|---|---|
| A | DATA AND TYPES | | |
| A1 | Typed Language | T | - LIS is a strongly typed language : the type of every variable or expression is determinable statically, at compile time.<br>- Note, however, that in implementation parts, alternate type interpretations may be used (see 7.2.3.2 qualified denotations). This is a necessity for the description of storage allocation algorithms among others. |
| A2 | Data types | P | - Integer, boolean, character, arrays and records are provided.<br>- Reals are not in the present LIS definition.<br>- Introducing reals in LIS should be simple |
| A3 | Precision | F | - See A2 (no reals...).<br>- See Tinman comment. |
| A4 | Fixed point numbers | F | - See A2 (no reals...).<br>- Fixed point numbers and compiler scaling raise issues of readability, reliability and compiler complexity. They better be left out : integers and reals should be enough. |
| A5 | Character data | P | - Characters are treated as a predefined discrete type. They may be used as other discrete types, e.g. as array indexes and case statement labels.<br>- On the other hand, LIS does not permit specification of the literal and order of characters.<br>- The requirement could be met by a definition of the character set in a standard prelude (study needed). |

| | REQUIREMENT | DEGREE | COMMENT |
|---|---|---|---|
| A6 | Arrays | T- | - Array component type and lower bound determinable at compile time, upper bound a entry to array allocation scope.<br>- Restriction : one dimension.<br>- Allowing more than one dimension is simpl (syntax would be as in Pascal). |
| A7 | Records | T | - LIS plexes have his capability.<br>- See Tinman comment. |
| B | OPERATIONS | | |
| B1 | Assignment and Reference | T | - Assignment and reference are defined for all simple and plex types.<br>- Assignment of arrays are denoted as slic assignments, e.g. A(FIRST..LAST):=B(FIRST..LAST) |
| B2 | Equivalence | T | - Note : this may be difficult to implement efficiently for records containing "hole |
| B3 | Relationals | P | - True except for symbolic types.<br>- Extension is straightforward however it much preferable for types defined by enumeration to be unordered by default. (See Tinman comment). |
| E4 | Arithmetic operations | P | - True but for real division and exponentiation.<br>- See A2 (no reals...). |
| B5 | Truncation and rounding | F | - No truncation and rounding since no real (see A2). |

| REQUIREMENT | | DEGREE | COMMENT |
|---|---|---|---|
| B6 | Boolean operations | P | - Operators include "and", "or", "not", and "xor".<br>- I strongly disagree with short circuit evaluation (see Tinman comment).<br>- A much preferable alternative is to let Boolean operators be ordinary operators, and have a control operation ANDTHEN : IF I <= 7 ANDTHEN A(I) >X THEN ... |
| B7 | Scalar operations | P | - Array assignments are permitted and denoted as slice assignments (see B1).<br>- Other scalar operations on slices are not presently possible but extension should be easy.<br>- The second part of the requirement is mor complex to satisfy (how do you mechanical ly detect identity of logical structure). Packing is a poor justification for the requirement : we can assume that record individual fields will be accessed only i the unpacked form. Hence only the logical structure of the latter will be ever described. |
| B8 | Type conversion | T | - No implicit conversions.<br>- Explicit integer/character conversions exist.<br>- Similar conversions should be introduced when reals are put into the language. |
| B9 | Changes in numeric representation | T | - No explicit conversion required for numerical ranges. |
| B10 | I/O operations | T | - This requirement is satisfied in LIS without additional language features. An input/output package has been defined as a partition, using type definitions and interfaces to the operating system primitives. |

| | REQUIREMENT | DEGREE | COMMENT |
|---|---|---|---|
| B11 | Power set operations | T | - Union ("or"), intersection ("and"), symmetric difference ("xor"), complement ("not"), and the element predicate "in" exist for powersets of enumeration types. |
| C | EXPRESSIONS AND PARAMETERS | | |
| C1 | Side-effects | N.A. | - Not applicable to LIS since functions hav no side-effects. Side-effects are "statement-like" rather than "expression-like", hence possible side-effects should be given as statements. (See Tinman comment).<br>- Again disagreement with boolean operators considered as control operations. |
| C2 | Operand structure | T | - Four levels of operator hierarchy :<br>(1) unary operators,<br>(2) multiplying operators,<br>(3) adding operators,<br>(4) relational operators. |
| C3 | Expressions permitted | T | - Expressions may appear in all places wher variables and constants may. |
| C4 | Constant expressions | T | - Constant expressions involving usual operators are evaluated at compilation time.<br>- See Tinman comment for limitations in current level of compiler technology. |
| C5 | Consistent parameter rules | T | - Parameter assignments obey the same rules as ordinary assignments. |
| C6 | Type agreement in parameters | T | - Formal and actual agree in type.<br>- Size and subscript range for array parameters is passed as a "row" (see Tinman comment). |

| REQUIREMENT | | DEGREE | COMMENT |
|---|---|---|---|
| C7 | Formal parameter kinds | P | - LIS provides for value, result, and value result parameters.<br>- Value-result is superior to reference in case of exceptions since the result assignment is only performed when returning<br>- Whereas exception handling is a valuable goal for a language, exception handling parameters represent too much complexity.<br>- Procedure parameters are not provided in LIS. They could be added if really needed This need is not clear however (a case statement containing alternate calls is a substitute). Moreover, formal procedures tend to require a more complex run-time organization even where they are not used |
| C8 | Formal parameter specification | F | - Generic procedures cannot be considered a state of the art. Note that the form suggested by this requirement calls for replication of procedure bodies when dealin with objects of different size.<br>- The complexity of code replications cannc be mastered if we consider separate compilation. Since this is far more important, the requirement of optional formal parame ter specification should be abandoned. |
| C9 | Variable number of parameters | F | - The complexity of the mechanism, as well as the limited field of application (especially with the restriction of having the same type) suggest deletion of this requirement.<br>Note : neither C8 nor C9 is state of the art. They could remain as suggestions in the (improbable) event that someone finds a simple solution applicable to the reall interesting cases : read and write. |

| | REQUIREMENT | DEGREE | COMMENT |
|---|---|---|---|
| D | VARIABLES, LITERALS AND CONSTANTS | | |
| D1 | Constant value identifiers | T+ | - Extended to dynamic constants (à la Algol 68). For instance, the upper bound of a dynamic array is a dynamic constant. |
| D2 | Numeric literals | T+ | - LIS also provide notations for literal aggregates. |
| D3 | Initial values of variables | T | - Variables may be initialized as part of their declaration. |
| D4 | Numeric range and step size | P | - Numeric and symbolic ranges exist.<br>- No fixed point variables. (See A4). |
| D5 | Variable types | T | - Arrays of records containing arrays of records ... *permitted*. |
| D6 | Pointer variables | T | - LIS has reference variables.<br>- References are type restricted.<br>- access restricted references exist : ref constant. |
| E | DEFINITION FACILITIES | | |
| E1 | User defined types | P | LIS offers two forms of definition facilities :<br>- partitions are groups of semantically related data, type and action definitions,<br>- plexes with functional attributes offer a capability for defining data and associated operations similar to that of the Simula class, but without the associated run-time costs,<br>- however LIS does not permit definition of new infix operators (neither does Simula) Operator definitions complicate syntactic error recovery, and, in general, the compilation process. Are they really worth the price ? (See Tinman comment). |

| | REQUIREMENT | DEGREE | COMMENT |
|---|---|---|---|
| E2 | Consistent use of types | T- | - Type definitions are entirely processed at compile-time and LIS allows full specification of their internal representation. Hence there is no penalty for using user defined types.<br>- Pushing this requirement to its extreme requires operator overloading and also an ability to define the syntax of literals of user defined types. In other words if double integer is defined by extension, you should be able to define a double integer literal with a syntax which is not necessarily that of aggregate literals, since you do not want to consider double integers as aggregates. This clearly show that this extreme is not in the state of the art. |
| E3 | No default declaration | T | - No default declaration in LIS (except those of the predefined types : boolean, character, ...). |
| E4 | Can extend existing operators | F | - See Tinman comment : is it worth the price ? |
| E5 | Type definitions | T | - Satisfied by plexes with functional attributes (analogous to Simula classes). |
| E6 | Data defining mechanisms | T | - The mechanisms listed in the requirement all exist in LIS. |
| E7 | No free union or subset types | T | - Symbolic ranges exist but are not considered as types |
| E8 | Type initialization | P | - Initialization and finalization may be performed in the bodies of the user supplied actions "Allocate" and "Free".<br>- On the other hand no initializations are defined directly with the type.<br>- Defining default initialization values fo plexe attributes is simple if limited to compile time expressions. Going beyond asks for some degree of compiler complexity as shown by Simula compilers.<br>- The whole requirement seems to contradict D3 (no default initial values) and, as a consequence, E2 (built-in and user defined types indistinguishable). |

| | REQUIREMENT | DEGREE | COMMENT |
|---|---|---|---|
| F | SCOPES AND LIBRARIES | | |
| F1 | Separate allocation and access allowed | T | - Scope of allocation is determined by the declarations of partitions and segments.<br>- Scope of access (visibility) is a subset of the scope of allocation determined explicitly by a context specification (e.g. "use data P,Q;" if access to P and Q is requested). |
| F2 | Limiting access scope | T- | - In partitions this is made possible by their division into two parts :<br>. a data partition containing the visible part,<br>. a program partition containing the hidden part.<br>- In plexes, attributes which are only accessible by functional attributes are declared as "private".<br>- On the other hand, LIS offers no renaming capability. This is clearly a text editor function. It has no place in a language. |
| F3 | Compile-time scope determination | T | - Scope is determined at compile-time.<br>- In addition, the "unique visibility rule" forbids redeclaration of an identifier at an inner level within a program module. Redefinitions are extremely error-prone and inhibit correctness proofs : you may prove properties of the wrong variable. |
| F4 | Libraries available | T | - LIS partitions are packages. For instance it is possible to define an I/O partition a partition containing mathematical functions, ...<br>- Partitions are separately compiled and may be put in libraries. |

| | REQUIREMENT | DEGREE | COMMENT |
|---|---|---|---|
| F5 | Library contents | T | - The mechanism for libraries exist (see F4<br>- Communication with routines written in other languages is possible using an interface specification. |
| F6 | Libraries and compools indistinguishable | T- | - Compools are available in the form of an "include" facility.<br>- Both partitions and compools may contain elements definable in the language. Partitions however, are more structured. I believe that the requirement should be understood broadly (you should have the same capabilities in both). Requiring them to be totally indistinguishable is certainly not a need. |
| F7 | Standard library definitions | T | - In LIS this does not require any additional language feature :<br>(a) the language offers a machine independent notation for describing machine dependent capabilities,<br>(b) a group of such interface definitions may be provided as a separately compiled partition. |
| G | CONTROL STRUCTURES | | |
| G1 | Kinds of control structures | P | - Sequential, conditional, iterative and recursive control are provided.<br>- Neither exception handling nor parallel processing and asynchronous interrupt handling are part of the LIS present definition. |
| G2 | The go to | P | - LIS offers limited forms of specialized gotos called "exits" and "returns".<br>- The above forms should be sufficient. |

| REQUIREMENT | DEGREE | COMMENT |
|---|---|---|
| G3 Conditional control | T | - If statements.<br>- Case statements.<br>- Note that given "exit" statements, the equivalent of Zahn's device may be writte as follows :<br><br>loop<br>  if event1 then<br>  statementlist1 ;<br>  exit ;<br>  end ;<br>  if event2 then<br>  statementlist2 ;<br>  exit ;<br>  end ;<br>  statementlist0 ;<br>repeat ;<br><br>This example is certainly at least as readable as :<br><br>loop until event1 or event2 ;<br>  statementlist0 ;<br>repeat ;<br>case<br>  <event1> : statementlist1 ;<br>  <event2> : statementlist2 ;<br>end ;<br><br>Hence no additional construct is require<br><br>In general, exotic control structures should be avoided ; they tend to be misi terpreted by programmers and are then re ponsible for reliability breakdowns : (i our example, for instance it is not clea with Zahn's form which statement list wi be executed if both events happen). |

| | REQUIREMENT | DEGREE | COMMENT |
|---|---|---|---|
| G4 | Iterative control | T | - The iteration condition may appear before after or anywhere in the loop (exit if condition ;). |
| G5 | Routines | T | - Both recursive and non-recursive are provided.<br>- The restriction suggested in the requirement should be an implementation restriction (see Tinman comment) for machines without base registers. |
| G6 | Parallel processing | F | - Study needed for the extension. |
| G7 | Exception handling | F | - Study needed for the extension. |
| G8 | Synchronization and Real Time | F | - Study needed for the extension. |
| H | SYNTAX AND COMMENT CONVENTIONS | | |
| H1 | General characteristics | T | - The LIS syntax is bounded context (3,1). |
| H2 | No syntax extensions | T | - Strictly no syntax extension. |
| H3 | Source character set | T | - Alphabetic : A to Z<br>- Numeric : 0 to 9<br>- Special : .<+\|*();-/,>:@$=#' and underscore |
| H4 | Identifiers and literals | P | - Break character for identifiers is the underscore.<br>- Strings may be broken in consecutive string elements such as 'first element' 'and' 'last element'.<br>- Presently no break character for integer literals : extension is easy (use underscore). |

| | REQUIREMENT | DEGREE | COMMENT |
|---|---|---|---|
| H5 | Lexical units and lines | T | - No continuation over lines.<br>- Strings longer than a line must be broken into string elements. |
| H6 | Key words | T | - Keywords are reserved. |
| H7 | Comment conventions | P | - LIS comments start with a $ and end with a $.<br>- Although multiple line comments are thus permitted, no danger results since the compiler prints a $ in the margin of multiple line comments.<br>- Scattered small comments are poor programming style and should be discouraged. The useful comments are really the long paragraphs stating facts. Hence, end-of-line termination does not contribute to adoption of a good programming style. |
| H8 | Unmatched parentheses | T | - No unmatched parentheses.<br>- All compound forms are well bracketted. |
| H9 | Uniform referents | T | - Uniform referents were one of the LIS design goals (see Trondheim conference article). |
| H10 | Consistency of meaning | T | - LIS satisfies the requirement. |
| I | DEFAULT, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS | | |
| I1 | No defaults in program logic | T | - No defaults in program logic. |
| I2 | Object representation specifications optional | T | - The compiler adopts a default implementation for all objects for which an implementation specification is not explicitly provided. |

| | REQUIREMENT | DEGREE | COMMENT |
|---|---|---|---|
| I3 | Compile time variables | T | - This is a special case of declarable compile time constant.<br>- Standard attributes are another form. |
| I4 | Conditional compilation | T | - This is a special case of dead code elimination and is presently treated by the LI optimizer.<br>- An interesting extension (not treated by the existing LIS compiler) could be made to declarative cases (variants) when the selector is a compile time constant. |
| I5 | Simple base language | T | (This requirement seems to duplicate the requirements on extension facilities - unless I misunderstand). |
| I6 | Translator restrictions | F | - Not in present definition.<br>- Easy to satisfy<br>Note : for highly symbolic programs the limit on the number of identifiers should be a few thousands rather than a few hundred a is common in untyped languages (statistics difficult to extrapolate). |
| I7 | Object machine restrictions | T | - No object environment restrictions. |
| J | EFFICIENT OBJECT REPRESENTATION AND MACHINE DEPENDENCIES | | |
| J1 | Efficient object code | T | - No cost imposed for unused generality.<br>Note : some features, such as procedure parameters have been left out just for that reason : if you have formal procedures, then the linkage informations maintained for all procedures may have to be more complex. |

| | REQUIREMENT | DEGREE | COMMENT |
|---|---|---|---|
| J2 | Optimization do not change program effect | T | - However, for incorrect programs, optimization may change the point where the run time error occurs. |
| J3 | Machine language insertions | T | - LIS interfaces provide an encapsulated form for machine code insertions and for the specification of non standard linkage conventions. |
| J4 | Object representation specification | T | - This is possible with plex implementation specifications.<br>- These specifications are distinct from the logical description (the type declaration) and are optional.<br>Note : this requirement seems to duplicat I2 (object representation specifications) |
| J5 | Open and closed routine calls | P | - Open routine capability is offered for machine language insertions.<br>- The LIS compiler opens routine which are called only once, or which are smaller than their calling sequence on the machin considered.<br>- Extending open capability to separately compiled routines seems rather complex.<br>- Shouldn't the choice be left to the translator in all cases. |

## 2. BEYOND THE TINMAN REQUIREMENTS

This section lists some areas of language design where LIS offers facilities which go beyond what is asked by the Tinman set of requirements.

We believe that these facilities should be present in a language claiming to represent the state of the art of the late seventies.

### 2.1. Separate compilation

It is clearly within the state of the art to provide separate compilation of modules and at the sometime permit type checking across separately compiled modules. Hence separate compilation should be made as an imperative requirement. This requirement is clearly, more important than many other language requirements (remember that when separate compilation is not provided, people resort to patching !). Hence in case of possible conflicts (such as parameterized types) separate compilation should be retained as the dominant requirement.

### 2.2. Standard attributes

When needed it should be possible to denote any property known by the compiler using notations of the language. Standard attributes are the answer to this requirement. They go much beyond the environment enquiries of Algol 68.

As an example, a machine code insertion, may need to operate on the size of a variable or of a procedure stack frame. This size may be denoted by a standard attribute of the variable or procedure.

Clearly we may want to restrict the contexts where some of the standard attributes may appear.

## 2.3. Non positional syntax

Non positional syntax (using a keyword notation for parameter passing, aggregates, ...) improves readability and hence reliability. It is consistent with the statement that "readability is more important than writeability". It may be used to "localize" the information needed for the understanding of programs (e.g. parameter passing mode).

## 2.4. Paragraphing

This is a necessity for the maintenance of large programs. It protects the professional program reader from the style accidents of individual program writers. It permits a more efficient reading since the conventions are not constantly changing.

Designing a language for standard paragraphing will bring syntax simplifications. It eliminates the necessity of finding exotic keywords for ending if, case, and other groups : the best way to indicate the end of any group is undoubtedly to have a paragrapher actually draw a vertical bar between the first and last keywords delimiting the group :

```
if condition then
|   statement ;
|   ...
|   statement ;
end ;
```

## 2.5. Two level language

It is important to be able to separate the logical description of entities from the description of the way they should be implemented. Some of the Tinman requirements point already in this direction (e.g. I2 and J4).

Exploiting the full logic of this attitude leads to grouping all implementation specifications in a textually separate implementation part.

The advantages are :

- a clear isolation of what is needed for the understanding of the program logic and of what is needed for its realization,
- a clear isolation of parts which are more likely to be affected by a transfer on a different machine,
- an ability to try alternate implementations in order to improve the space and/or time requirements of a program, without changing the program logic.

## 3. SUMMARY

This section summarizes the results of the comparative evaluation in table form.

Clearly some of the requirements are more easily met than others. For instance, LIS either fails or partially satisfies requirements A2, A3, B4 and B5 for a single, easily repairable reason which is the absence of reals in the present definition.

In order to give more perspective to this evaluation we have stratified the answers in five categories :

(1) LIS : This summarizes the degree of compliance to the requirements of the present LIS definition.

(2) EASY : A star indicates which requirements can be satisfied in a straight-forward manner.

(3) STUDY : A star indicates a requirement not presently satisfied and for which
(a) an extension should be made,
(b) a study will be needed in order to decide how the extension should be done.

(4) DOUBTS : A star indicates a requirement which is considered to be very debatable. Depending on the case, this may apply to part of or to the whole of the requirement.

(5) DISAGREE : A star indicates a requirement on which there is a strong disagreement, either because it is not state of the art, or because it is believed to be either too complex or dangerous. I strongly recommend that these requirements be dropped, C8 and C9 being possibly left as suggestions.

The language that could evolve from LIS in an attempt to satisfy the Tinman requirements should have a degree of compliance lying somewhere between those of the columns "study" and "doubt" Note also that these two columns differ more in degree than in the nature of the requirements addressed.

I do believe that any language satisfying the Tinman in this range, should be a significant improvement compared to the existing mostly use languages.

| REQUIREMENT | LIS | EASY | STUDY | DOUBTS | DISAGREE | nature of doubt or disagreement |
|---|---|---|---|---|---|---|
| **A. DATA AND TYPES** | | | | | | |
| A1 Typed Language | T | | | | | |
| A2 Data Types | P | * | | | | |
| A3 Precision | F | | * | | | |
| A4 Fixed Point Numbers | F | | | | * | |
| A5 Character Data | P | | * | | | |
| A6 Arrays | T | | | | | |
| A7 Records | T | | | | | |
| **B. OPERATIONS** | | | | | | |
| B1 Assignment and Reference | T | | | | | |
| B2 Equivalence | T | | | | | |
| B3 Relationals | P | * | | | | |
| B4 Arithmetic Operations | P | * | | | | |
| B5 Truncation and Rounding | F | * | | | | |
| B6 Boolean Operations | P | | | | * | short circuit |
| B7 Scalar Operations | P | | | * | | "identical logical structure" |
| B8 Type Conversion | T | | | | | |
| B9 Changes in Numeric Representation | T | | | | | |
| B10 I/O Operations | T | | | | | |
| B11 Power Set Operations | T | | | | | |
| **C. EXPRESSIONS AND PARAMETERS** | | | | | | |
| C1 Side Effects | N.A. | | | | | |
| C2 Operand Structure | T | | | | | |
| C3 Expressions Permitted | T | | | | | |
| C4 Constant Expressions - | T | | | | | |
| C5 Consistent Parameter Rules | T | | | | | |
| C6 Type Agreement in Parameters | T | | | | | |
| C7 Formal Parameter Kinds | P | | | *1 | *2 | 1- procedure parameters<br>2- exception parameters |
| C8 Formal Parameter Specifications | F | | | | * | |
| C9 Variable Numbers of Parameters | F | | | | * | |

| REQUIREMENT | LIS | EASY | STUDY | DOUBTS | DISAGREE | nature of doubt or disagreement |
|---|---|---|---|---|---|---|
| D. VARIABLES, LITERALS AND CONSTANTS | | | | | | |
| D1 Constant Value Identifiers | T | | | | | |
| D2 Numeric Literals | T | | | | | |
| D3 Initial Values of Variables | T | | | | | |
| D4 Numeric Range and Step Size | P | | | | * | step size |
| D5 Variable Types | T | | | | | |
| D6 Pointer Variables | T | | | | | |
| E. DEFINITION FACILITIES | | | | | | |
| E1 User Definitions Possible | P | | | * | | operators |
| E2 Consistent Use of Types | T | | | | | |
| E3 No Default Declarations | T | | | | | |
| E4 Can Extend Existing Operators | F | | | * | | operator extension |
| E5 Type Definitions | T | | | | | |
| E6 Data Defining Mechanisms | T | | | | | |
| E7 No Free Union or Subset Tupes | T | | | | | |
| E8 Type Initialization | P | | * | | | |
| F. SCOPES AND LIBRARIES | | | | | | |
| F1 Separate Allocation and Access Allowed | T | | | | | |
| F2 Limiting Access Scope | T | | | | | |
| F3 Compile Time Scope Determination | T | | | | | |
| F4 Libraries Available | T | | | | | |
| F5 Library Contents | T | | | | | |
| F6 Libraries and Compools Indistinguishable | T | | | | | |
| F7 Standard Library Definitions | T | | | | | |

| REQUIREMENT | LIS | EASY | STUDY | DOUBTS | DISAGREE | nature of doubt or disagreement |
|---|---|---|---|---|---|---|
| G. CONTROL STRUCTURES | | | | | | |
| G1 Kinds of Control Structures | P | | * | | | |
| G2 The Go To | P | | | * | | no need of full go to |
| G3 Conditional Control | T | | | | | |
| G4 Iterative Control | T | | | | | |
| G5 Routines | T | | | | | |
| G6 Parallel Processing | F | | * | | | |
| G7 Exception Handling | F | | * | | | |
| G8 Synchronization and Real Time | F | | * | | | |
| H. SYNTAX AND COMMENT CONVENTIONS | | | | | | |
| H1 General Characteristics | T | | | | | |
| H2 No Syntax Extensions | T | | | | | |
| H3 Source Character Set | T | | | | | |
| H4 Identifiers and Literals | P | * | | | | |
| H5 Lexical Units and Lines | T | | | | | |
| H6 Key Words | T | | | | | |
| H7 Comment Conventions | P | | | * | | single line comments |
| H8 Unmatched Parentheses | T | | | | | |
| H9 Uniform Referent Notation | T | | | | | |
| H10 Consistency of Meaning | T | | | | | |
| I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS | | | | | | |
| I1 No Defaults in Program Logic | T | | | | | |
| I2 Object Representation Specifications Optional | T | | | | | |
| I3 Compile Time Variables | T | | | | | |
| I4 Conditional Compilation | T | | | | | |
| I5 Simple Base Language | T | | | | | |
| I6 Translator Restrictions | F | * | | | | |
| I7 Object Machine Restrictions | T | | | | | |

| REQUIREMENT | LIS | EASY | STUDY | DOUBTS | DISAGREE | nature of doubt or disagreement |
|---|---|---|---|---|---|---|
| J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES | | | | | | |
| J1 Efficient Object Code | T | | | | | |
| J2 Optimizations Do Not Change Program Effect | T | | | | | |
| J3 Machine Language Insertions | T | | | | | |
| J4 Object Representation Specifications | T | | | | | |
| J5 Open and Closed Routine Calls | P | | | * | | open separately compiled routines |
| SUMMARY | | | | | | |
| T | 53 | 59 | 66 | 72 | | |
| P | 15 | 11 | 8 | 3 | | |
| F | 10 | 8 | 4 | 3 | | |

# PEARL versus TINMAN

A comparative evaluation of PEARL, the Process and Experiment Automation Realtime Language, against the DOD-TINMAN-Requirements for High Order Computer Programming Languages (june 1976)

Contents: 1. Introduction
2. The Formalized Evaluation of PEARL

(Status: September, 1976)

Presented by:
Dr. Tomas Martin
Chairman, PEARL Development Boa
- PDV Project -

## 1. Introduction

This document contains information on PEARL in the formalized format as required for the machine automated evaluation over the ARPANET.

(The following individuals contributed to this document: Elzer, Ghassemi, Holleczek, Lindstedt, Martin, Muehlhahn, Pelz, Prester, Reh, Roessler.)

The comparison has been made both against the PEARL Basic Subset and full PEARL.

The PEARL Subset for Avionic Applications (Language Description June 1976) which has been distributed earlier is rather close to the PEARL Basic Subset.

For additional information about PEARL refer to the informal "Information about PEARL (Status: Oct. 1976)".

2. Formalized Evaluation (12 pages)

Reference Language: P E A R L

| Needed Characteristic, Requirement | | Degree of Compliance | Comments |
|---|---|---|---|
| RefNo | Title | T = complete<br>P = partial<br>F = Fail<br>U = Unknown | |
| IV.A. | Data and Types | | |
| 1 | Typed Language | T | |
| 2 | Data Types | P | no fixed point data, Full-PEARL contains this characteristic |
| 3 | Precision | T | |
| 4 | Fixed Point Numbers | F | no fixed point data |
| 5 | Character Data | F | no enumeration type |
| 6 | Arrays | P | all dimensions should be known at compile time |
| 7 | Records | F | |
| IV.B. | Operations | | |
| 1 | Assignment and Reference | T | |
| 2 | Equivalence | P | possible only for simple data types such as integer, real bit, character, clock, duration |
| 3 | Relationals | P | no enumeration type |
| 4 | Arithmetic Operations | P | the user has no access to the remainder of integer division it is accessible by modulo operation |
| 5 | Truncation and Rounding | T | |

Reference Language: P E A R L

| Needed Characteristic, Requirement | | Degree of Compliance | Comments |
|---|---|---|---|
| RefNo | Title | (T,P,F,U) | |
| 6 | Boolean Operations | T | no short circuit mode |
| 7 | Scalar Operations | F | Full-PEARL contains this characteristic |
| 8 | Type Conversion | P | some implicit conversion by assignment |
| 9 | Changes in Numeric Representation | T | |
| 10 | I/O Operations | P | device variable in Full-PEARL |
| 11 | Power Set Operations | F | |
| IV.C. | Expressions and Parameters | | |
| 1 | Side Effects | T | |
| 2 | Operand Structure | T | |
| 3 | Expressions Permitted | T | |
| 4 | Constant Expressions | T | evaluation depends on the implementation |
| 5 | Consistent Parameter Rules | T | |
| 6 | Type Agreement in Parameters | P | dimension of array should be known at compile time, but not in Full-PEARL |

Reference Language: P E A R L

| Needed Characteristic, Requirement | | Degree of Compliance | Comments |
|---|---|---|---|
| RefNo | Title | (T,P,F,U) | |
| IV.C. | | | |
| 7 | Formal Parameter Kinds | P | exception condition can not be the formal parameter<br>Full-PEARL permits procedures as formal parameter |
| 8 | Formal Parameter Specifications | P | not optional, but necessary |
| 9 | Variable Numbers of Parameters | F | the number of parameters should be always the same |
| IV.D. | Variables, Literals and Constants | | |
| 1 | Constant Value Identifiers | T | |
| 2 | Numeric Literals | T | |
| 3 | Initial Values of Variables | T | |
| 4 | Numeric Range and Step Size | P | no fixed point, this in Full-PEARL |
| 5 | Variable Types | P | no enumeration |
| 6 | Pointer Variables | F | no pointer, this in Full-PEARL |

E. DEFINITION FACILITIES

Reference Language: P E A R L



| Needed Characteristic, Requirement | | Degree of Compliance | Comments |
|---|---|---|---|
| RefNo | Title | (T,P,F,U) | |
| E1 | User Definitions Possible | F | Only provided in the full language specification |
| E2 | Consistent Use of Types | F | Indistinguishable in the full language |
| E3 | No Default Declaration | T | |
| E4 | Can Extend Existing Operators | F | Only provided in the full language specification |
| E5 | Type Definitions | F | |
| E6 | Data Defining Mechanisms | F | |
| E7 | No Free Union or Subset Types | T | |
| E8 | Type Initialization | F | |

Reference Language: P E A R L

## F. SCOPES AND LIBRARIES

| Needed Characteristic, Requirement | | Degree of Compliance | Comments |
|---|---|---|---|
| RefNo | Title | (T,P,F,U) | |
| F1 | Separate Allocation and Access Allowed | F | There are no allocation facilities in PEARL since there was an agreement that the dangers of letting the applications programmer interfere with the O.S. are too great |
| F2 | Limiting Access Scope | T | Program entities in a 'PEARL-Module' may be declared as GLOBAL and thus be accessed from outside in principle. But it is necessary to specify those elements in other modules in addition when they are used therein. |
| F3 | Compile Time Scope Determiniation | T | |
| F4 | Libraries Available | P | There is only a small set of arithmetic functions already specified to be contained in a standard library. It seems to be a question of management to extend the scope in a suitable manner. |
| F5 | Library Contents | T | Supported by module structure (esp. in full PEARL) |
| F6 | Libraries and Compools Indistinguishable | P | There is no explicit COMPOOL-facility in PEARL. If a PEARL-module is used in a way that is equivalent to a COMPOOL there are some slight differences to the use of a library. |
| F7 | Standard Library Definitions | T | Besides a full set of statements for formatted I/O there are facilities for transfer of binary and/or control information (TAKE/SEND with P-option), which allow handling of special devices. The connections to/between any application dependent system components may be described in the so called system description, which supports the formulation of the respective transfer mechanisms. |

G. CONTROL STRUCTURES

Reference Language: P E A R L

| Needed Characteristic, Requirement | | Degree of Compliance | Comments |
|---|---|---|---|
| Ref No | Title | (T,P,F,U) | |
| G1 | Kinds of Control Structures | P (T) | PEARL does contain structured control mechanisms, like:<br>case-statement<br>while-branch of repeat statement<br>if-then-else-clause<br>It contains an extensive set of control mechanisms for (pseudo) parallel processing, but these were not yet designed according to the principles of structured programming.<br>PEARL is capable of handling asynchronous interrupts and time schedules. Exceptions are treated in the form of signals. |
| G2 | The Go To | P | PEARL contains a goto-statement which obeys the usual ALGOL-rules for its range. It is not allowed to enter or to leave a parallel process by means of a 'goto'. |
| G3 | Conditional Control | P | The else clause may be omitted in PEARL but since the if-statement must always be closed by the keyword 'FIN' there cannot be any ambiguity and so the choice is explicit. There is no selection based on union subtypes. |
| G4 | Iterative Control | P | Most requirements are fulfilled in PEARL but it is only possible to specify the termination conditions at the head of the loop (except by use of 'goto'-statement). |
| G5 | Routines | F (P) | PEARL does not provide for recursive procedures in order to avoid the necessity of dynamic storage allocation, and since the readability of recursive programs was regarded not to be very convenient. On the other hand it is allowed to optionally specify procedures as 'reentrant' in order to make them shareable between parallel processes, but to force the user to remember that this may require some extra runtime cost. |
| G6 | Parallel Processing | T | |
| G7 | Exception Handling | T | Handled by signals; the user may react in a way he wishes to any error or exception which can be recognized by the operating system. |
| G8 | Synchronization and Real Time | T | Speciality of PEARL. One exception: interrupts are treated like schedules rather than like exceptions. |

H. SYNTAX AND COMMENT CONVENTIONS

Reference Language: P E A R L

| Needed Characteristic, Requirement | | Degree of Compliance | Comments |
|---|---|---|---|
| RefNo | Title | (T,P,F,U) | |
| H1 | General Characteristics | P | Req. fulfilled except : Significance of the first 6 characters of identifiers. |
| H2 | No Syntax Extensions | T | |
| H3 | Source Character Set | T | |
| H4 | Identifiers and Literals | P | R.f.e.: There is no break character for use internal to identifiers and literals. |
| H5 | Lexical Units and Lines | P | R.f.e.: Lexical units may cross lines (e.g. long character strings) |
| H6 | Key Words | P | R.f.e.: In a context, where a monadic operator occurs (by use of a keyword) also the call of a user procedure (by using an identifier) is possible. Keywords are reserved. |
| H7 | Comment Conventions | P | Comment is not automatically terminated by 'end-of-line', because there does not exist the 'end-of-line' on language definition level (according to H1.) |
| H8 | Unmatched Parentheses | T | |
| H9 | Uniform Referent Notation | T | |
| H10 | Consistency of Meaning | P | The END key word is matching not only BEGIN<br>The FIN " " " " " " IF |

## I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

Reference Language: P E A R L

| Needed Characteristic, Requirement | | Degree of Compliance | Comments |
|---|---|---|---|
| RefNo | Title | (T,P,F,U) | |
| I1 | No Defaults in Program Logic | T | |
| I2 | Object Representation Specifications Optional | T | Reentrant/non-reentrant and resident/non-resident attributes |
| I3 | Compile Tune Variables | F | |
| I4 | Conditional Compilation | F | |
| I5 | Simple Base Language | T | |
| I6 | Translator Restrictions | T | This requirement is met by the PEARL-Basis-Subset, but not by full PEARL |
| I7 | Object Machine Restrictions | T | |

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

Reference Language: P E A R L

| Needed Characteristic, Requirement | | Degree of Compliance | Comments |
|---|---|---|---|
| RefNo | Title | (T,P,F,U) | |
| J1 | Efficient Object Code | - | Some requirements are not concerned with the language PEARL but with the compiler resp. the program environment. As there do not exist general requirements for PEARL compilers resp. the software environment some existing PEARL installations meet these requirements and others do not. Therefore these questions cannot be answered. |
| J2 | Optimizations Do Not Change Program Effect | T | |
| J3 | Machine Language Insertions | F | only by use of library procedures |
| J4 | Object Representation Specifications | F | |
| J5 | Open and Closed Routine Calls | F | |

K. PROGRAM ENVIRONMENT

Reference Language: P E A R L

| Needed Characteristic, Requirement | | Degree of Compliance | Comments |
|---|---|---|---|
| RefNo | Title | (T,P,F,U) | |
| K1 | Operationg System Not Required | F | PEARL requires an operationg system, the functions of which are defined by the language definition. |
| K2 | Program Assembly | T | |
| K3 | Software Development Tools | P | Test and operation aids are in development |
| K4 | Translator Options | - | See comment IV, J1 |
| K5 | Assertions and other Optional Specifications | F | |

## L. Translators

Reference Language: P E A R L

| Needed Characteristic, Requirement | | Degree of Compliance | Comments |
|---|---|---|---|
| RefNo | Title | (T,P,F,U) | |
| L1 | No Superset Implementations | - | See comment IV, J1 |
| L2 | No Subset Implementations | T | |
| L3 | Low-Cost Translation | - | See comment IV, J1 |
| L4 | Many Object Machines | - | " " " " |
| L5 | Self-Hosting Not Required | - | " " " " |
| L6 | Translator Checking Requ'd | - | " " " " |
| L7 | Diagnostic Massages | - | " " " " |
| L8 | Translator Internal Structure | T | |
| L9 | Self-Implementable Language | F | Not so far. |

M. LANGUAGE DEFINITION, STANDARDS AND CONTROL

Reference Language: P E A R L

| Needed Characteristic, Requirement | | Degree of Compliance | Comments |
|---|---|---|---|
| RefNo | Title | (T,P,F,U) | |
| M1 | Existing Language Features only | P | |
| M2 | Unambigious Definition | T | Formal semantic definition of PEARL is in production |
| M3 | Language Doc. Required | T | Complete documentation will be available in early 1977 |
| M4 | Control Agent Required | - | not applicable |
| M5 | Support Agent Required | - | |
| M6 | Library Standards and Support Required | - | |

# AN ASSESSMENT OF THE PROGRAMMING LANGUAGE CORAL 66 AGAINST THE U.S. DEPARTMENT OF DEFENSE TINMAN REQUIREMENTS

Software Sciences Ltd.,
Abbey House,
282-292 Farnborough Road,
Farnborough, Hampshire.
Telephone Farnborough 44321.
Telex 858248.

Reference F656

November 1976.

## INTRODUCTION

Section 1 of this document gives an overall assessment of CORAL 66 against the TINMAN and attempts to put CORAL in historical perspective. Section 2 then evaluates CORAL against each TINMAN requirement in turn. This section is structured in the same lines as the TINMAN document with evaluations against both the major and minor requirements. An attempt has been made to allow each section and subsection to stand on its own for reference purposes. Some duplication of information is therefore inevitable. The third section provides a graphical summary: it obviously represents a very coarse measure.

A full specification of CORAL 66 can be found in:

"Official Definition of CORAL 66" published by Her Majesty's Stationary Office, London, 1970.

The TINMAN requirements are given in an appendix.

## 1. OVERALL ASSESSMENT OF CORAL 66

### RATIONALE AND DEVELOPMENT

CORAL 66 was designed by members of the Royal Radar Establishment (RRE) which is a Ministry of Defence (MOD) research and development establishment. The language was not intended to compete with established languages in use on large scientific or commercial machines, but was designed for use in implementing real-time systems using small, dedicated computers. This application area had previously been programmed almost exclusively in assembly or machine code. The following design aims of CORAL were regarded as essential for a high level language to replace assembler in this area:

> the language must permit rapid compilation with minimal equipment (i.e. moderate core, no backing store, paper tape i/o);
>
> the language should give access to all available machine facilities;
>
> the efficiency and size of the object code should be comparable with that produced by an assembler and should not require elaborate hardware for efficient execution;
>
> it should be capable of implementation on a wide range of machines.

The Official Definition of CORAL 66 was published in 1970, at which time the language was adopted by the MOD as a standard for all military programming. The MOD informed all major computer manufacturers that in the selection of computers for defence projects, preference would be given to those offering CORAL 66. The Inter-Establishment Committee on Computer Applications (IECCA) representing the three services, the MOD and its procurement organisation and R.&D. was established. This committee administers a list of preferred computers for use in defence projects: the provision of a CORAL 66 compiler being a requirement

Cont'd.

for inclusion on this list.

In 1973 discussions were held between the MOD, the Department of Industry, the Software Houses Association, the Control and Automation Manufacturers Association, the National Computing Centre, nationalised industries and the Post Office to examine the case for a national standard language for real-time industrial and process control work. The result was the establishment of the CORAL 66 Group with the aim of promoting CORAL 66 as a real-time language standard funded by the Government. Control of the standard is now vested in a representative body of interested parties and is outside the direct control of any one person, establishment or company. CORAL is now in widespread use in many application areas including control of power stations, steel mills, radar systems, air traffic control systems, message switching and computer operating systems.

TECHNICAL DESCRIPTION

The language is based on Algol 60 with some features from JOVIAL and some from FORTRAN. The design was constrained to allow the possibility of efficient compilation, loading and execution. Compilation is in principle a one-pass process with identifiers required to be specified or defined in advance of their use to ensure the availability of all relevant information when required. The syntax of the language is transformable into one-track predictive form, to enable fast syntax analysers to be used. Dynamic store allocation is not required, nor is run-time checking of procedure entries. Except for diagnostic purposes, array bound checking is not required.

CORAL programs are divided into segments which are independantly compiled. Each segment is similar to an Algol 60 block and may contain procedures and blocks nested to arbitrary depth. Communicators are defined which specify those objects external to a segment which are accessible from within the segment. A common communicator is defined together with recommendations for library and other communicators.

Cont'd.

CORAL provides static allocation for all items outside recursive procedures.

Basic numeric types are provided for integer, fixed point and floating point values. Arrays may be one or two dimensional and must have constant bounds. 'Tables' may be declared and are effectively one-dimensional arrays of records. The elements of such a record may be of one of the basic numeric types, are accessible via identifiers defined with the table and may be held packed. Arithmetic and logical operators are defined and may be used between basic data types or expressions bracketed to any depth. A location operator may be used to determine the machine address of data. The inverse operation is provided and these enable list processing to be performed, although not with the benefits of type checking.

Procedures may return values or not. Their formal parameters must have a declared type which must be matched by the corresponding actual parameters. Data may be passed by value as in Algol 60, or by reference as in FORTRAN. Procedures, labels and switches may also be passed. Recursive procedures are provided, but must be explicitly declared as such.

CORAL does not define any facilities for parallel processing or real time synchronisation. This was to enable the language to be used on machines with a wide variety of supervisory software. At the time, the theory of parallel processing had not advanced to the stage that the choice of language facilities in this area was inevitable. The parameter mechanism is sufficiently powerful to enable the programmer to communicate with supervisory software via macro or procedure call. A code facility is provided which enables exploitation of all available machine facilities.

## CONCLUSION

CORAL 66 is the standard programming language in use in real-time applications in the U.K. It is a general purpose programming language designed for efficient compilation and execution with minimal equipment.

It is well engineered language using mid-sixties technology.



Cont'd.

As such it scored badly on those TINMAN requirements which are based on subsequent developments in compiler and programming technology, e.g. extensibility, type checking, parallel processing.

Despite these technological limitations, CORAL is in widespread use for real-time programming throughout the U.K. and has proved itself a worthwhile standard language.

## 2. DETAILED ASSESSMENT.

### A. DATA AND TYPES.

CORAL bears a strong family resemblance to Algol 60. Like that language it provides a fixed set of basic numeric data types and the ability to construct arrays whose elements are restricted to these basic types. Within these constraints CORAL provides consistent type checking. CORAL additionally allows the user to define "tables" which are a form of record, although their description is strongly biased towards their physical representation. As with arrays, the elements of tables are constrained to be of the basic numeric types. The table mechanism does not guarantee consistency of type checking: tables of different structures are regarded as interchangeable for many purposes within the language.

The basic data types provided are:

(a) integer: their range is that of the object environment

(b) fixed: fixed point numbers may be declared of differing ranges and accuracies, these latter quantities being expressed in terms of numbers of bits in the object representation. Scale factor management is performed by the compiler.

(c) floating: floating point numbers have a range and precision determined by the object machine.

Although characters and booleans are not provided, the language provides conversion operators between print characters and their integer values in the target machine; it also provides diadic logical operations for use between typed primaries - the $i^{th}$ bit of the result being a logical function of the $i^{th}$ bits of the object representation of the operands.

Arrays are restricted to one or two dimensions and are required to have compile-time fixed bounds. This is consistent with the CORAL design objective of eliminating the need for run-time storage management.

In summary, CORAL is a simple, non-extensible language oriented towards numeric computations and capable of compilation and execution on small machines of limited capability. In this context it is excellent and has conferred the benefit of high-level language programming in many areas previously the exclusive province of assembly or machine-code programming. Outside this context, its limitations have become apparent. Its fixed range of data types do not confer the benefits of data structuring and type checking provided by modern, extensible languages such as Algol 68 or Pascal.

A1. TYPED LANGUAGE

CORAL 66 requires explicit specification of the type of all variables, parameters and components of composite data structures. The type of expressions and operations can easily be derived from those of the variables involved. CORAL provides a fixed set of types: it does not allow the user to define his own types. MOREOVER it should be noted that CORAL also explicitly provides mechanisms for avoiding type checking: e.g. the 'overlay' facility may associate variables of different types with the same storage locations, the bits operator gives access to any bit slice within the machine representation of variables and the 'anonymous reference' facility gives access to any actual machine location: the contents of the location are taken to be of numeric type integer irrespective of any declaration which may have associated that location with some other type.

## A2. DATA TYPES

CORAL provides data types for floating point, fixed-point and integer variables. It provides one or two dimensional arrays whose elements may be of one of the above types. Records may be declared in the form of tables: see under requirement A7.

Note that CORAL does not provide either Boolean or Character as basic types, however, the language does provide Boolean and logical operations: see under requirement B6.

## A3. PRECISION

There is no explicit specification of the precision of floating point variables or operations: their precision is that of the execution environment.

## A4. FIXED POINT NUMBERS

This requirement is totally met by CORAL: fixed point variables are treated as exact quantities with a range and fractional step size specified by the user as part of the declaration. Scale factor management is done by the compiler. Integers are similarly treated as exact quantities.

## A5. CHARACTER DATA

CORAL does not include type definition by enumeration, nor does it include 'character' as a basic data type. The language assumes that each printing character has a unique integer representation within the object computer, dependent on some hardware or software convention. This value is represented in the source program by the literal operator,

e.g. literal (a)

has an integer value uniquely representative of 'a'. Layout characters are not permitted as arguments of literal. CORAL also defines the syntax of strings - i.e. any succession of characters (printing or layout) enclosed in quotation marks (string quotes). Strings may be used in the language wherever an 'unconditional expression' is permitted. The language definition recommends that procedures capable of selecting individual characters from a string should be designed so that characters are represented by the same integer values as for literal constants. The language definition does not further specify the operations permitted on strings.

## A6. ARRAYS

Arrays in CORAL can be of one or two dimensions: their type and bounds are explicitly specified at compile-time. (This derives from the design aim of the language of permitting compile-time allocation of all variables except those declared within recursive procedures.)

c.f. Requirement C6 for formal array parameters and the mechanism of passing arrays as parameters.

## A7. RECORDS

Records are provided in CORAL in the form of 'tables'. A table is effectively a one-dimensional array of 'table-entries': each entry being a structured collection of 'table-elements'. The table-elements are in turn basic numeric types (i.e. integer, floating or fixed-point variables). The declaration of a table defines the width of an entry (measured in computer words), the number of such entries and their structure. Note that the definition of the structure of a table entry is strongly oriented towards the physical layout of the structure, viz. each table element is defined by its name, its type and its physical location in terms of word and bit positions relative to the start of the table entry.

The elements may occupy full computer words or part computer words (although they may not overlap word boundaries), they may overlap and they need not altogether occupy the available space in the entry. Access to table element in the source program is made via the name of the element subscripted by the index of the required table entry. Since table elements may overlap one another, it is possible to provide alternative record structures. However, such alternative structures are not discriminated at run time (i.e. the compiler does not maintain a 'tag' field defining the current type of value held in the alternative structures). CORAL tables therefore permit inconsistent interpretation of types.

B. OPERATIONS

CORAL provides basic numeric data types and operations on them. These operations include assignment and reference, arithmetic operations, logical operations, identity and relational operations. Data structures are limited to arrays and tables ( a form of record). No operations apply to these structures as a whole, the user being required to program these operations in terms of operations on the individual elements.

CORAL does not define any input-output operations, files, etc. Such mechanisms are assumed to be provided by external supervisory software or to be implemented in CORAL as a set of procedures. Similarly, powersets are not explicitly provided, although the logical operations provided cater for the powerset operations.

Type conversions are performed implicitly in certain contexts according to a defined set of rules. A facility is provided to effect such conversions.

B1. ASSIGNMENT AND REFERENCE

Assignment and reference are defined for all built-in data types within CORAL. Note that CORAL does not permit user definition of types or assignment and does not provide union variables.

B2. EQUIVALENCE

CORAL provides boolean operators as follows:

$<$, $\leq$, $=$, $\geq$, $>$, $\neq$

These apply to any of the basic types or to expressions involving variables of these types: they do not however apply to array or structure variables but are restricted to use with their individual elements. Note that CORAL does not provide user defined types. The accuracy of equality between floating point variables is that of the execution environment.

B3. RELATIONALS

CORAL provides all six relational operators $<$, $\leq$, $=$, $\geq$, $>$, $\neq$, which apply to any of the basic data types. CORAL does not provide data type definitions via enumeration. See also under B2.

## B4. ARITHMETIC OPERATIONS

Built-in arithmetic operations are available for addition, subtraction, multiplication and division and negation. There is no exponentiation operator. The language definition does not discuss the type of the operands and result of the division operation. The accuracy of floating point operations is that of the execution environment.

## B5. TRUNCATION AND ROUNDING

This issue is not explicitly covered by the language definition, although in practice any reasonable implementation will meet this requirement fully.

## B6. BOOLEAN OPERATIONS

CORAL takes a distinctive view of the handling of boolean and logical values. There is no boolean type present in the language. Boolean expressions which affect the flow of control in the program (i.e. those appearing after the keywords "if" and "while")are known as 'conditions'. Conditions are made up of arithmetic comparisons (e.g. "a < b + c") connected by the boolean operators "and" and "or". The boolean operators have their usual meaning with "and" binding the most tightly and "or" being inclusive. Conditions are evaluated in short circuit mode - i.e. from left to right only as far as is necessary to determine their truth or falsity. Note that the boolean operator "not" is not available within conditions and that brackets are not permitted within conditions. This reflects their function of directing the flow of control.

In addition to the above, CORAL provides three diadic logical operators which are defined for use between typed primaries (note that the only types provided are various numeric types). The effect of these operations is implementation dependent to the extent that the object representation of data is not defined by the language. The i$^{th}$ bit of the result is a given logical function of the i$^{th}$ bit of the two operands, the result as a whole having a numeric type of integer. To avoid confusion with the boolean operations in 'conditions', a different terminology is used. The operators are:

Differ

| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

Union

| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

Mask

| | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

these correspond to 'not equivalent', 'inclusive or' and 'and' respectively. Expressions involving these logical operators are not evaluated in short-circuit mode.

B7. SCALAR OPERATIONS

CORAL does not meet this requirement as it does not provide any operations applicable to arrays or records as a whole. Assignment and other operations may only apply to the individual data elements of arrays and records.

B8. TYPE CONVERSION

CORAL does provide implicit type conversion between numeric types according to explicit rules for the evaluation of expressions, however it also provides a mechanism for explicitly controlling such conversions within expressions. For a detailed exposition, see CORAL official definition section 6.1.3. Note also that CORAL does not provide union types.

B9. CHANGES IN NUMERIC REPRESENTATION

CORAL does not require explicit conversion operations between numeric types or ranges. The language definition does not discuss exception conditions, therefore such exceptions as occur on truncation will be implementation dependent.

B10. I/O OPERATIONS

No input-output operations are defined within CORAL.

B11. POWER SET OPERATIONS

CORAL does not provide type definitions via powersets. It does, however, provide logical operations (difer, union, mask) which can be used to fabricate powerset operations. These are discussed under requirement B6.

## C. EXPRESSIONS AND PARAMETERS

CORAL provides a consistent set of syntax and semantic rules for expressions. The form of expressions allowed is not restricted in special contexts such as subscript expressions. The semantic rules include definition of the evaluation order of operands. The language does not permit constant expressions to appear in place of constants.

Parameter kinds allow for the transmission of data values, of references to data variables, arrays and structures, and for the passing of procedures, labels and switches. Assignment of actual to formal parameters follows a set of rules consistent with the assignment of expressions to variables. Formal parameters may be used within their scope as if they were local variables. The type of formal parameters must be specified in the procedure heading and must be matched by the actual parameters. It should be noted, however, that the dimensionality and bounds of arrays are not checked, and that no distinction is made between records of different structures. No provision is made for variable numbers of parameters.

C1. SIDE EFFECTS

The official definition of CORAL requires that functions (i.e. procedures delivering results) occuring in expressions be evaluated in the order in which they appear when the expression is read from left to right, regardless of brackets. This is of course subject to the qualification that the arguments of a function must necessarily be evaluated before the function itself is evaluated. Apart from functions, the compiler is free to choose the order of evaluation of expressions Since functions are the only participants of expressions capable of producing side effects, this requirement is fully satisfied.

C2. OPERAND STRUCTURE

CORAL includes 8 levels of operator precedence and the use of brackets to achieve a required association between operators and operands. It is not possible for the user to modify the precedence of existing operators nor to introduce new operator precedence rules.

C3. EXPRESSIONS PERMITTED

This requirement is totally satisfied by CORAL.

C4. CONSTANT EXPRESSIONS

CORAL does not meet this requirement.

C5. CONSISTENT PARAMETER RULES

This requirement is totally met by CORAL.

## C6. TYPE AGREEMENT IN PARAMETERS

The parameter passing mechanism in CORAL is effectively an assignment of the actual parameter to the formal parameter. The formal and actual parameters must agree in type within the normal constraints of assignment. (c.f. requirements B8, B9 and C5).

It should be noted that formal array parameters do not explicitly state the dimensionality nor the bounds of the required actual parameters. The actual parameter is passed as the address of the (possibly conceptual) zero$^{th}$ element: the parameter passing mechanism does not include implicit passing of the array bounds. The formal parameter may be indexed within the body of the procedure to give access to the elements of the actual array.

## C7. FORMAL PARAMETER KINDS

Formal parameter classes within CORAL are as follows:

for data:

a) "value" parameters which represent the value of the actual parameter at the time of call. These are declared as, e.g. value integer vi.

b) "location" parameters which pass a reference to the actual parameter. These are declared as e.g. location integer li.
The corresponding actual parameter must be a variable of the correct type.

for procedures:

the specification of the formal parameter defines the number and type of its formal parameters: the corresponding actual parameter must match this specification.

for places:

labels and switches may be passed as parameters.

It should be noted that arrays and records can only be passed by location.

As there is no exception handling mechanism in CORAL, there is no corresponding formal parameter class.

## C8. FORMAL PARAMETER SPECIFICATION

Type specification of formal parameters is not optional within CORAL.

## C9. VARIABLE NUMBERS OF PARAMETERS

CORAL does not provide for variable numbers of parameters.

## D. VARIABLES, LITERALS AND CONSTANTS

CORAL provides a syntax for constant values of built-in data types, although such values cannot be associated with identifiers. Variables may be initialised as part of their declaration, this initialisation taking place as the program is initially loaded (CORAL supports compile time allocation of storage for variables).

Fixed point variables must have range and stepsize specified as part of their declaration. The range of integers and the range and precision of floating point variables is determined by the object environment.

CORAL is an Algol 60 style language designed in the mid 1960's for programming small computers in the real-time and process control field. As such it provides a small set of basic numeric types and arrays of such numeric elements. It also provides "tables", a form of record whose elements must also be one of the basic numeric types. The set of data types available is therefore limited.

As mentioned above, CORAL provides compile time allocation of storage for variables. Moreover, the language definition lays down certain criteria for the allocation strategy. Operations are provided in the language for converting between variable names and their machine addresses, and for using such values for manipulating the contents of variables. This enables pointer values to be manipulated and used, although not within the security of compile-time type checking.

## D1. CONSTANT VALUE IDENTIFIERS

CORAL does not provide the facility of associating constant values with identifiers except via macro definitions.

## D2. NUMERIC LITERALS

The CORAL syntax defines the representation of constants for all built-in data types within source programs. The representation of constants in data is not defined.

## D3. INITIAL VALUES OF VARIABLES

CORAL permits the initial value of variables to be specified as part of their declaration. CORAL is designed to allow static, compile-time storage allocation for variables. The initialisation facility reflects this: only variables declared at segment level (i.e. outermost block level) or within segment level non-recursive procedures are eligible for initialisation. This initialisation is performed at program load time, and not dynamically at entry to the scope of the variable. There is no run-time facility for testing for initialisation.

## D4. NUMERIC RANGE AND STEP SIZE

CORAL does not permit specification of the range of integer or floating point variables. For fixed point variables the range and step size are both specified in terms of the numbers of bits in the object representation. These specifications are interpreted as part of the type information: note, however, that the compiler will insert implicit conversion operations between different numeric ranges in appropriate contexts.
c.f. B8 and B9.

## D5. VARIABLE TYPES

CORAL provides a fixed set of numeric data types and does not permit their extension by the definition of new types. Data structures are therefore limited to those provided in the language definition, i.e. individual numeric variables, one or two dimensional arrays and records whose elements are individual numeric items.

## D6. POINTER VARIABLES

CORAL does not provide explicit pointer types for variables. Instead it provides two facilities 'anonymous references' and the 'location' operator which may be used to map between objects in the program and machine addresses.

An 'anonymous reference' takes the form:

[Index]

Where Index is any integer-valued expression. It may be used in any context where the name of a variable is permitted and causes the machine location whose address is given by 'Index' to be treated as an integer variable.

The location operator is the inverse operator and takes the form:

location (Wordreference)

Where Wordreference is either the name of a variable or an element of an array or an anonymous reference. The operator gives the machine address of the Wordreference as an integer.

These operators permit the manipulation of pointers in CORAL, however, they do not provide any security of typechecking. Moreover they assume that machine addresses may be represented as a single dimension linear sequence.

## E. DEFINITION FACILITIES

CORAL matches up to the TINMAN requirements very poorly in this area. CORAL is of the same vintage and style as Algol 60 and does not provide extensible facilities. Of the various subrequirements in this section of the TINMAN, only that precluding default declarations is met.

### E1. USER DEFINITIONS POSSIBLE

CORAL 66 is a fixed language providing a small number of numeric data types. It is not possible for the user to extend these with his own type definitions. Operations can be defined solely as procedure definitions; their formal parameters are restricted to the inbuilt data types.

### E2. CONSISTENT USE OF TYPES

This requirement is not met in that user-defined types are not permitted.

### E3. NO DEFAULT DECLARATIONS

This requirement is fully satisfied: CORAL does not provide default declarations.

### E4. CAN EXTEND EXISTING OPERATORS

This requirement is not met: operators may only be defined in the form of procedures.

E5. TYPE DEFINITIONS

This requirement is not met: CORAL does not permit user type definitions.

E6. DATA DEFINING MECHANISMS

This requirement is not met: CORAL does not permit user type definitions.

E7. NO FREE UNION OR SUBSET TYPES

CORAL does not meet this requirement. Although it does not provide type definition by explicit union, it does provide implicit free union facilities. For instance the 'overlay' facility causes different variables, which may be of different types, to share the same storage locations (viz FORTRAN equivalence). Similar effects may be achieved within table (i.e. record) definitions.

Range definitions are only permitted with fixed-point numeric variables: the range definition is then construed as part of the type. See B9.

E8. TYPE INITIALIZATION

This requirement is not met: CORAL does not permit user type definitions.

## F. SCOPES AND LIBRARIES

CORAL provides the familiar Algol 60 block structured scope rules. These are entirely processed and checked at compile-time. The storage allocation scheme allows compile-time allocation for all variables except those declared within recursive procedures.

CORAL defines a set of "communicators" which make available objects declared external to a segment (i.e. unit of compilation). These communicators include common, library, external and absolute. The common communicator is fully defined, others are merely described in outline, their details being implementation dependent. The language definition encourages consistency between the various forms of communicator.

## F1. SEPARATE ALLOCATION AND ACCESS ALLOWED

This requirement is fully satisfied. CORAL provides block structured access to objects combined with static, compile time allocation for all variables except those declared within the body of recursive procedures. The access scope may never be wider than the allocation scope.

F2. LIMITING ACCESS SCOPE

This requirement is not met: the access scope of objects is block structured similar to Algol 60.

F3. COMPILE-TIME SCOPE DETERMINATION

This requirement is totally satisfied.

F4. LIBRARIES AVAILABLE

The Official Definition of CORAL provides an outline description of various 'communicators' which make separately defined objects available to the source program. This includes a library communicator which makes available library procedures and data objects. The detailed form of this communicator and the contents of the libraries are not defined.

F5. LIBRARY CONTENTS

As mentioned under F4, the contents of libraries is not discussed in the language definition, although provision is made in the language for such libraries. The form in which the libraries is held is not defined.

## F6. LIBRARIES AND COMPOOLS INDISTINGUISHABLE

CORAL provides 'communicators' for libraries (see F4 and F5) and common, although the detailed form that these should take is not defined.

## F7. STANDARD LIBRARY DEFINITIONS

The Official Definition of CORAL makes no recommendation for forms that operating system calls should take, nor of their semantics.

## G. CONTROL STRUCTURES

CORAL defines control structures for sequential, iterative, conditional and recursive control within a single sequential program.

The design of the language dates from the mid-1960's, at which stage there was little agreement on the theory and structure of concurrent programming. The language definition therefore states that external, supervisory software is required when using CORAL in real-time applications. Although stating that communication with such supervisory software is made via procedure or macro calls, no details of such interfaces are specified.

### G1. KINDS OF CONTROL STRUCTURES

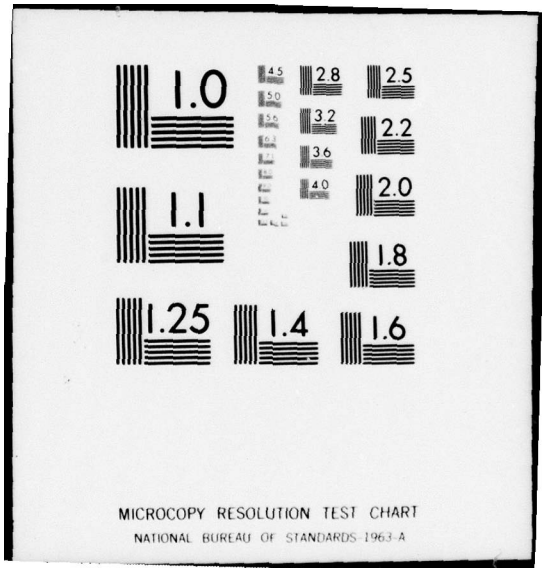
CORAL provides control structures for sequential, conditional and recursive control. It does not define any structures for parallel processing, exception handling or a synchronous interrupt handling. The control structures provided are described under requirements G3, G4 and G5.

AD-A037 634 LANGUAGE EVALUATION COORDINATING COMMITTEE F/G 9/2
REPORT TO THE HIGH ORDER LANGUAGE WORKING GROUP (HOLWG),(U)
JAN 77 S AMOROSO, P WEGNER, D MORRIS, D WHITE

UNCLASSIFIED NL

28 OF 28
ADA037634

END
DATE FILMED
4-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

## G2. THE GO TO

CORAL provides a goto statement which causes the transfer of control to an explicit label within scope. Note that the label is not required to be within the current block but may be within an enclosing block subject to normal rules of block structured access. It should also be noted that CORAL also provides switches and formal parameters for both labels and switches.

## G3. CONDITIONAL CONTROL

CORAL provides conditional statement of the form

if Condition then Consequence else Alternative

where 'else Alternative' is optional, 'Alternative' may be any statement, including a further Conditional statement: 'Consequence' may be a restricted form of statement, which does not include a further'Conditional statement'. This avoids the "Dangling else" problem and makes the Conditional statement fully partitioned. If it is required to place a further Conditional statement following 'then', the enclosed Conditional must be surrounded by begin - end brackets.

No form of case clause is provided.

## G4. ITERATIVE CONTROL

The iterative control structure provided by CORAL is effectively that of Algol 60 except that the expressions used in the clause

Expression step Expression until Expression

are evaluated once only on entry to the statement. The for-statement imposes the use of a loop control variable and requires that the termination condition appear at the head of the loop (see TINMAN requirement G1). The loop control variable is global to the iterative control.

## G5. ROUTINES

CORAL provides both recursive and non-recursive routines. Recursive routines must be explicitly defined as such by substituting the language word 'recursive' for 'procedure'. Procedures can be nested to any depth, including the declaration of recursive procedures one within another.

## G6. PARALLEL PROCESSING

The CORAL definition does not specify any mechanism for parallel processing. It is expected that such mechanisms would be provided by supervisory software accessed via procedure or macro calls. No definition of these facilities is given.

## G7. EXCEPTION HANDLING

No exception handling mechanism is defined within CORAL.

## G8. SYNCHRONISATION AND REAL TIME

No mechanism is defined in CORAL for synchronisation or time delays. It is assumed that such mechanisms are provided by supervisory software accessed via procedure or macro calls. No definition of these facilities is given.

## H. SYNTAX AND COMMENT CONVENTIONS

CORAL matches up to the TINMAN requirements well in this area. CORAL has a simple, consistent syntax which is transformable into one-track predictive form which enables fast syntax analysis. The language is free format with a strong family resemblance to Algol 60. The parse of a program is not modified by declarations contained within it. The only departure from the TINMAN requirements is that CORAL defines three comment conventions and does not possess a uniform referent notation.

### H1. GENERAL CHARACTERISTICS

This requirement is totally satisfied.

### H2. NO SYNTAX EXTENSIONS

This requirement is totally satisfied. It is not possible to modify operator precedence or in other ways to modify the program parse.

### H3. SOURCE CHARACTER SET

This requirement is fully satisfied. The language definition recommends standard conversions between language symbols (e.g. begin, for) and their representation in restricted character sets.

## H4. IDENTIFIERS AND LITERALS

CORAL provides formation rules for identifiers and literals. Layout characters are ignored (except within string quotes) and space may therefore be used as break characters without acting as terminators of lexical items.

## H5. LEXICAL UNITS AND LINES

The dividing line between syntactic and lexical levels is somewhat arbitrary and is usually drawn on purely pragmatic grounds given the target syntax and the input character set. The syntax of CORAL is such that this requirement may be fully satisfied, although the existing lexical formation rules are not sufficient to guarantee this.

## H6. KEYWORDS

This requirement is fully satisfied. Keywords are distinguished from identifiers as they are composed from a different alphabet (represented by enclosure in primes within restricted character sets).

H7. COMMENT CONVENTIONS

This requirement is not met by CORAL. CORAL defines three comment conventions, viz.

a) A Comment statement may be placed anywhere that statements or declarations may appear. It takes the form of the keyword comment followed by text and terminated by a semicolon. The text may not include a semicolon.

b) Bracketed comment may appear following a semicolon of the program. It takes the form of text enclosed in round brackets. The text may contain brackets provided they are matched.

c) Comment in the form of an identifier may appear following the symbol end.

H8. UNMATCHED PARENTHESES

This requirement is fully satisfied.

H9. UNIFORM REFERENT NOTATION

This requirement is not met: access to arrays takes the form

arrayname [Indices]

whereas procedure calls take the form

procedurename (parameters).

H10. CONSISTENCY OF MEANING

This requirement is fully satisfied. In particular, equality and assignment are represented by = and := respectively.

## I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

CORAL was designed in the mid 1960's as a general purpose programming language for use particularly with small computers in the process control and real-time field. The Official Definition states:"In such fields of application, some debasement of high-level language ideals is acceptable if, in return, there is a worthwhile gain in speed of compilation with minimal equipment and in efficiency of object code. The need for a language which takes these requirements into account, even though it may not be fully machine independent, is widely felt in industrial and military work". The language therefore includes certain facilities which are compiler or target machine dependent. No conditional compilation feature is specified.

## II. NO DEFAULTS IN PROGRAM LOGIC

CORAL includes various areas where the effect of the program is dependent on the object environment. For example the logical operators are defined to operate on the individual bits of the object representation of values. These areas are explicitly mentioned in the language definition. Subject to this qualification, however, those program properties not fixed within the language definition must be explicitly defined in the program.

## I2. OBJECT REPRESENTATION SPECIFICATIONS OPTIONAL

No mechanisms are defined whereby the programmer can direct the compilation strategy. Moreover data definitions are not entirely independent of the object representation, e.g. (a) table (= record) definitions define the physical representation (see A7), (b) compilers are explicitly required to use consecutive storage locations for items in composite declarations such as "int i, j, k;".

## I3. COMPILE TIME VARIABLES

No compile time variables are defined.

## I4. CONDITIONAL COMPILATION

There is no defined conditional compilation facility.

## I5. SIMPLE BASE LANGUAGE

CORAL is a small, simple non-extensible language. As such it is effectively its own base language although this becomes a somewhat meaningless concept.

## I6. TRANSLATOR RESTRICTIONS

The CORAL definition makes some translator limits explicit, recommends other limits and leaves others undefined. Examples of these are respectively:

a) array dimensionality limited to 2

b) compilers may disregard all but the first 12 printing characters in identifiers

c) number of nested parentheses in expressions undefined.

## I7. OBJECT MACHINE RESTRICTION

The Official Definition of CORAL does not define object environment limits, nor does it define the compilers handling of such limits.

## J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

The possibility of efficient compilation and object code was a design aim of CORAL. For example run-time storage allocation is not required except for recursive procedures and these are required to be explicitly declared as such. CORAL is in effect its own simple base language in which the user does not pay the costs of unneeded generality.

The language does not distinguish between the logical definition and the object definition of data. Also no mechanisms are defined for directing the compilation strategy (e.g. open vs. closed routines).

### J1. EFFICIENT OBJECT CODE

CORAL was specifically designed to allow the production of efficient object code. For example features such as dynamic storage allocation and access are excluded from the language; recursive procedures are required to be explicitly declared as such in the interests of efficiency. The language was also specifically designed to allow fast compilation, e.g. the syntax may be parsed by a one-track predictive analyser, identifiers must be declared before they are used etc.

## J2. OPTIMISATIONS DO NOT CHANGE PROGRAM EFFECT

The CORAL definition does not discuss optimisations.

## J3. MACHINE LANGUAGE INSERTIONS

CORAL provides a codestatement which may appear anywhere that a statement is allowed.

This takes the form:

code begin Codesequence end

The detailed form of the codesequence is not defined but is required to enable the programmer to exploit all available facilities of the object environment. Access to any identifier in scope is also required.

There is no conditional compilation feature (see I3 and I4): code statements may be freely interspersed with other statements. It is recommended that code statements provide for the inclusion of nested CORAL text.

## J4. OBJECT REPRESENTATION SPECIFICATIONS

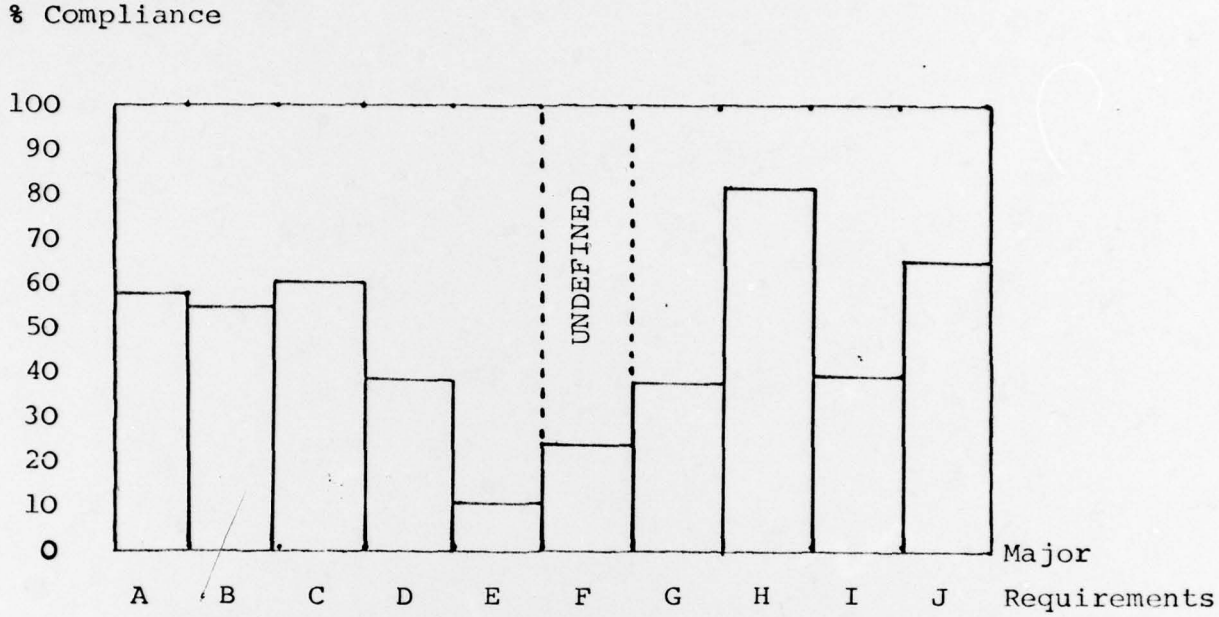
CORAL does not distinguish between the logical description of data structures and their representation. The Tabledefinition is strongly oriented towards the physical representation of data: table (i.e. record) elements are specified in terms of physical location and size as well as name and type.

## J5. OPEN AND CLOSED ROUTINE CALLS

CORAL does not allow the programmer to specify whether routines should have an open or closed implementation. The compiler is free to choose the implementation it determines to be optimal.

It should be noted that CORAL includes a macro facility which is frequently used to define code routines. Macro and procedure calls are syntactically equivalent, although no type checking is available with macros.

## 3. SUMMARY CHART




A 56% DATA AND TYPES

B 50% OPERATIONS

C 59% EXPRESSIONS AND PARAMETERS

D 38% VARIABLES, LITERALS AND CONSTANTS

E 10% DEFINITION FACILITIES

F - SCOPES AND LIBRARIES

G 37% CONTROL STRUCTURES

H 81% SYNTAX AND COMMENT CONVENTIONS

I 39% DEFAULTS, CONDITIONAL COMALATION AND LANGUAGE RESTRICTIONS

J 68% EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

## APPENDIX: THE TINMAN REQUIREMENTS

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will bedeterminable at compile time and unalterable at run time. The language will require that the type of each variable, and component of composite data structures be explicitly specified in the source programs.

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e. composite data structures with labeled components of heterogeneous type) as type generators.

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.

A5. Character sets will be treated as any other enumeration type.

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration.

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

B6. The built-in Boolean operations will include "and," "or," "not" and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode.

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversions operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

B10. The base language will provide operations allowing programs to interact with files, channels or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

B11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recongized.

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contributes to ease of learning.

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

C8. Specification of the type, range, precision, dimension, scale and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

D1. The user will have the ability to associate constant values of any type with identifiers.

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

D5. The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.[/]

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

E1. The user of the language will be able to define new data types and operations within programs.

E2. The "use" of defined types will be indistinguishable from built-in types.

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

E4. The user will be able, within the source language, to extend existing operators to new data types.

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

F1. The language will allow the user to distinguish between scope of allocation and scope of access.

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

F3. The scope of identifiers will be wholly determined at compile time.

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

H1. The source language will be free format with an explicit statement delimiter, willallow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences.

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

H8. The language will not permit unmatched parentheses of any kind.

H9. There will be a uniform referent notation.

H10. No language defined symbols appearing in the same context will have essentially different meanings.

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

I3. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

I4. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

I5. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency safety, or understandability of the language.

I6. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

I7. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

J2. Any optimizations performed by the translator will not change the effect of the program.

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.